UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR DE LEGANÉS
INGENIERIA INFORMÁTICA

PROYECTO FIN DE CARRERA

SOLUCIONES OPEN-SOURCE DE INTEROPERATIVIDAD ENTRE JAVA Y CLI

DIRECTOR

DAVID EXPÓSITO SINGH

AUTOR:

JUAN MANUEL FERNÁNDEZ RIBAO

Febrero, 2006

Índice

1. Introducción.

1.1. Introducción a las plataformas de desarrollo basadas en máquinas virtuales.

Una plataforma de desarrollo es el entorno común en el cual se desenvuelve la programación de un grupo definido de aplicaciones. Comúnmente se encuentra relacionada directamente a un sistema operativo, sin embargo, también es posible encontrarlas ligadas a una familia de lenguajes de programación o a una Interfaz de programación de aplicaciones (API por sus siglas en inglés).

Una máquina virtual es un programa nativo, es decir, ejecutable de una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial, generado generalmente por un compilador de lenguaje de alto nivel. El código binario no es un lenguaje de alto nivel, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico.

El principal objetivo con el que se comenzaron a desarrollar las máquinas virtuales era aportar portabilidad a un tipo de aplicaciones escritas para funcionar sobre ellas. De manera con diferentes implementaciones de una máquina virtual para diferentes arquitecturas, un programa escrito en un Windows® para esa máquina virtual puede ser interpretado en un entorno Linux®. Tan solo es necesario disponer de una implementación de dicha máquina virtual en ese entorno.

La recolección de basura suele ser una característica que incorporan las máquinas virtuales. La recolección de basura mejora la productividad de los programadores ya que les permite centrarse en lo que es el problema en si que esté tratando, en lugar andar continuamente preocupándose de liberar memoria. Las aplicaciones hechas para plataformas con recolección de basura son en teoría más estables al carecer de los errores causados por la liberación de memoria.

Las máquinas virtuales se sitúan entre las aplicaciones y el sistema operativo, este tipo de aislamiento puede ser usado para limitar el conjunto de operaciones que un programa puede realizar de acuerdo a unas determinadas políticas de seguridad. El uso de políticas de seguridad dificulta a los posibles virus que se pudieran camuflar en aplicaciones de este tipo su labor al limitar el acceso a los recursos y configuraciones del sistema.

En el apartado 2 de esta documentación se puede encontrar una introducción a tres plataformas de este tipo: CLI, Java™ y Parrot.

1.2. Interoperatividad.

En toda organización es común tener aplicaciones heredadas en diversas plataformas. Estas aplicaciones han sido escritas hace tiempo y

encontrar profesionales expertos en ellas para corregir fallos y añadir nuevas funcionalidades es cada vez más complicado. Además de todo el tamaño de estas aplicaciones suele ser demasiado grande para realizar una migración completa de las mismas a una nueva plataforma. A pesar de todos los inconvenientes todavía funcionan relativamente bien y no resulta rentable cambiarlas. Sin embargo nuestro deseo es usar las nuevas herramientas de programación y las nuevas plataformas de desarrollo más productivas.

Puede ser que no tengamos que elegir entre las ventajas de continuar usando las aplicaciones heredadas e implementar nueva funcionalidad en plataformas de desarrollo actuales. Lo único que necesitamos es que las aplicaciones heredadas sean capaces de hablar con las nuevas a través de un lenguaje común. Esta solución se conoce como interoperatividad.

Según el ISO/IEC 2382 Information Technology Vocabulary interoperatividad es la capacidad para comunicarse, ejecutar programas o transferir datos entre varías unidades funcionales, de una manera que suponga que el usuario necesite poco o ningún conocimiento de las características únicas de esas unidades.

También es importante considerar la interoperatividad aunque no tengamos aplicaciones heredadas. Es conveniente diseñar considerando la Interoperatividad ya que:

- Permite aprovechar y sustituir activos existentes
  - Integración de sistemas heredados
  - Los que no serán estratégicos en el futuro
  - Implica prepararse para la migración y/o sustitución
- Posibilita realizar planes de adopción
  - Puesto que la migración no es un proceso "de un día"
- Reduce el coste total de propiedad (TCO) de los nuevos proyectos
  - Apoyándose en los beneficios anteriores
- Reduce riesgos de inversión
  - Si se opta por usar una nueva plataforma de desarrollo y esta fracasa comercialmente no afectará a todos los programas de la organización.

1.3. Motivación general.

Actualmente las plataformas de desarrollo más pujantes en el mercado son Java™ y el .Net Framework. El .Net Framework es propiedad de Microsoft®, compañía líder en sistemas operativos y programas de ofimática. El .Net Framework está basado en un estándar llamado CLI en el que Novell está desarrollando la plataforma de fuente abierta Mono. Java™ es una tecnología propiedad de Sun™, pero además está apoyada por compañías importantes entre las que podemos destacar a IBM®, Oracle® y Bea®.

En Internet se pueden encontrar numerosas soluciones de interoperatividad entre estas plataformas que facilitan a los desarrolladores la posibilidad de escoger una u otra plataforma para una nueva aplicación y comunicarse con las ya existentes con independencia de si estas se han realizado en Java™, .Net o Mono.

Un catálogo o índice de estas soluciones es una herramienta de gran utilidad para todos aquellos desarrolladores que se vean necesitados de comunicar aplicaciones escritas en alguna de las plataformas antes mencionadas y conocer rápidamente la utilidad que más le conviene.

Lamentablemente los catálogos que se encuentran publicados por Internet o tratan de demostrar las bondades de una determinada solución sobre otras con un fin comercial o contienen un número de soluciones pequeño y poco variado.

Este proyecto pretende ofrecer un catalogo varías soluciones que explique las ventajas y las limitaciones de cada una acompañado de ejemplos. Se pretende que este catálogo permita a los desarrolladores hacerse una idea rápida de la solución que más le conviene de las tratadas en él para posteriormente profundizar en ella.

1.4. Objetivos del proyecto.

En este proyecto se tratará de exponer un conjunto de soluciones de interoperatividad entre las plataformas Java™ y CLI.

Este proyecto se va a centrar en aquellas soluciones de interoperatividad gratuitas y que tengan el código fuente disponible en Internet. El proyecto se centra en soluciones gratuitas porque se pretende demostrar que la interoperatividad no tiene porque acarrear costes extras en los proyectos. Tener el código fuente disponible resulta útil para tener un mayor control de la aplicación, es indispensable en los casos en que sea necesario recompilar o adaptar alguna librería. Se han excluido aquellas soluciones que aún siendo gratuitas y teniendo el código disponible impongan algún tipo de licencia a los programas que hagan uso de ellas, se pretende que todo lo tratado sirva en programas comerciales.

En las plataformas Java™ y CLI es posible usar simplemente sockets o ficheros a través de los cuales implementar métodos ac-hoc de comunicación a bajo nivel entre aplicaciones. Estas soluciones tampoco serán tratadas ya que según la definición de del apartado 1.2. no son soluciones de interoperatividad ya que obligan al programador un amplio conocimiento de la manera en que se realiza esa comunicación.

Las soluciones tratadas aquí proporcionarán al menos una abstracción de llamada a procedimiento, paso del estado de un objeto o envío de un mensaje entre aplicaciones de las citadas plataformas.

En este proyecto se usará C# en la plataforma CLI y Java™ para la plataforma del mismo nombre En todas las soluciones no serán necesario otros lenguajes de programación o de especificación de interfaces. En el caso de que se necesiten archivos en lenguajes adicionales como WSDL o

OMG™ IDL estos se generarán siempre a partir de otros escritos en C# o Java™.

Como complemento al catálogo se han realizado tres librerías con el objetivo de ofrecer soluciones adicionales a las que pueden encontrarse por la Internet. En cada uno de esos desarrollos se ha partido de una solución ya existente en Java™ y se ha portado a C#.

No se pretende tratar todos los métodos de interoperatividad posibles, pero si de ofrecer un conjunto razonablemente amplio y variado de soluciones.

Uno de los principales objetivos de este proyecto es que todos los métodos vistos en él sean sencillos de programar, usar y aprovechables en aplicaciones sea cuál sea su licencia.

1.5. Estructura de este documento.

El documento se compone de 7 apartados y dos apéndices.

1.	**Introducción.**	Explicación de los objetivos del proyecto y del contenido del documento.
2.	**Plataformas de desarrollo basadas en máquinas virtuales.**	Breve descripción de las plataformas de desarrollo Java™, CLI y Parrot.
3.	**Interoperatividad CLI/Java.**	Catálogo de soluciones de interoperatividad entre las plataformas Java™ y CLI.
4.	**Desarrollo de librerías MOM sobre CLI.**	Exposición del diseño de tres librerías desarrolladas en este proyecto usadas en el apartado anterior.
5.	**Evaluación de la eficiencia del canal de eventos.**	Comparativa de la eficiencia del la implementación del servicio de eventos de CORBA™ en Jacorb y de la realizada en este proyecto.
6.	**Desarrollo de herramienta de monitorización de un Cluster.**	Ejemplo de desarrollo de una herramienta de monitorización de un cluster usando el servicio de eventos de CORBA™.
7.	**Conclusiones.**	Conclusiones obtenidas de la realización del proyecto.
8.	**Trabajo futuro.**	Propuestas para nuevos desarrollos de interoperatividad entre Java™ y CLI.

Apéndices:

A)	**Bibliografía.**	Libros usados en el proyecto.
B)	**Direcciones de Internet.**	Direcciones de páginas web consultadas en el proyecto.

1.6. Acrónimos.

ADO	ActiveX® Data Objects
CLS	Common Language Specification
CLI	Common Language Infrastructure
COM	Component Object Model
CORBA®	Common Object Request Broker Architecture
CTS	Common Type System
DTD	Document Type Definition
ECMA	European Computer Manufacturer's Association
EJB	Enterprise JavaBeans™
ESB	Enterprise Service Bus
GAC	Global Assembly Cache
GIMP	GNU Image Manipulation Program
GNOME	GNU Object Model Environment
GNU	GNU's Not UNIX
GPL	General Public License
GTK	Gimp ToolKit
GUID	Globally Unique Identifier
IBM®	Industries Business Machine
IOR	Interoperable Object Reference
ISO	International Organization for Standardization
J2EE™	Java™ 2 Enterprise Edition
J2ME™	Java™ 2 Micro Edition
J2SE™	Java™ 2 Standard Edition
JACOB	Java™ COm Bridge
JBI	Java™ Business Integration
JCP™	Java™ Community Process
JDBC™	Java™ Database Connectivity
JMS	Java™ Message Service
JNDI™	Java™ Naming and Directory Interface
JNI	Java™ Native Interface
JSR	Java™ Specification Request
JVM	Java™ Virtual Machine
LGPL	Lesser General Public License
LINQ	Language Integrated Query
MIT	Massachusetts Institute of Technology
MSDN	Microsoft Developer Network
MSMQ	Microsoft Message Queuing
OMG®	Object Management Group
ORB	Object Request Broker
PASM	Parrot Assembly
PAST	Parrot Abstract Syntax Tree
PBC	Parrot Bytecode
PIR	Parrot Intermediate Representation
PHP	PHP: Hypertext Preprocessor
PMO	Program Management Office

POA	Portable Object Adapter
POI	Poor Obfuscation Implementation
RMI	Remote Method Invocation
SOAP	Simple Object Access Protocol
STOMP	Streaming Text Orientated Messaging Protocol
SWIG	Simplified Wrapper and Interface Generator
SWT	Standard Widget Toolkit
TCL	Tool Command Language
UJSR	Umbrella Java™ Specification Request
UMO	Universal Message Object
URL	Uniform Resource Locator
UUID	Universal Unique IDentifier
VES	Virtual Execution System
W3C	World Wide Web Consortium
WCF	Windows Communication Foundation
WSE	Web Services Enhancements
XML	eXtensible Markup Languag
XMLRPC	XML Remote Procedure Call
XPCOM	Cross-platform Component Object Model
XSLT	Extensible Stylesheet Language Transformations

2. Plataformas de desarrollo basadas en máquinas virtuales.

2.1. CLI.

La infraestructura de lenguaje común (CLI) es una especificación de la ECMA de código ejecutable y de un entorno de ejecución virtual (VES) en el cual ejecutarlo.

La ECMA (European Computer Manufacturers Association) es una asociación dedicada a la estandarización de sistemas de información. Entre sus más importantes especificaciones además de CLI están los lenguajes de programación C#, JavaScript y Eiffel.

En el corazón de CLI tenemos un sistema de tipos unificados, el sistema de tipos común (CTS) que es compartido por todos los lenguajes, herramientas y por el propio CLI. Es el modelo que define las reglas que el CLI sigue cuando se declaran, usan y manejan tipos.

El CTS establece un entorno de trabajo que permite una integración entre distintos lenguajes de programación, *tipados* y buen desempeño en la ejecución de código.

La arquitectura de CLI comprende:

- Lenguaje de tipos común (CTS) – El CTS proporciona un sistema de tipos amplio para soportar los tipos y operaciones de varios lenguajes de programación.

- Metadata – El CLI usa *metadata* para describir y referenciar los tipos definidos por el CTS. Los *metadatas* son guardados de manera independiente a los lenguajes de programación. Esto es así porque proporcionan un mecanismo de intercambio entres distintas herramientas, como por ejemplo compiladores y depuradores, que manipulan los programas, también entre esas herramientas y el VES.

- Especificación de lenguaje común (CLS) – El CLS es un acuerdo entre desarrolladores de lenguajes y de *frameworks*, (es decir, la librería de clases). Especifica un subconjunto del CTS y un grupo de convenciones de uso. Los lenguajes proporcionan a sus usuarios la mejor manera de acceder al *framework* implementando como poco esas partes del CTS que son partes del CLS. De igual manera, los *framework* serán usados más ampliamente si exponen sus características (como pueden ser clases, interfaces, métodos y campos) usando solamente tipos del CLS y siguiendo los acuerdos del CLS.

- Sistema de ejecución virtual (VES)- El VES implementa y refuerza el modelo del CTS. El VES es responsable de cargar y ejecutar los programas escritos en CLI. Proporcionan los servicios para ejecutar código manipulado y datos, usando los *metadatos* para conectar módulos generados por separado en tiempo de ejecución.

C# es un lenguaje de programación desarrollado junto a la plataforma CLI. Aunque las implementaciones de C# normalmente usan CLI como librería y entorno de ejecución la especificación del lenguaje afirma que esto no es necesario.

CLI y C# han sido pensados para trabajar en común. Por este motivo el programador puede usar todas las características de CLI con C# de manera natural y obtener así una buena productividad. Al tratarse de estándares tenemos en el mercado varías implementaciones de ambos. Este ha sido el motivo por el que hemos seleccionado C# para la realización del proyecto.

Actualmente ECMA se encuentra redactando C++/CLI, una extensión de C++ para usar CLI. Esta extensión incluye nuevas palabras reservadas, clases, excepciones, espacios de nombres, además de recolector de basura.

2.2. Java™.

Java™ es una plataforma de software desarrollada por Sun Microsystems™, de tal manera que los programas creados en ella puedan ejecutarse sin cambios en diferentes tipos de arquitecturas y dispositivos computacionales.

La *plataforma Java*™ consta de las siguientes partes:

- El lenguaje de programación, mismo.
- La máquina virtual de Java™ o JRE, que permite la portabilidad en ejecución.
- El API Java, una biblioteca estándar para el lenguaje.

Originalmente llamado OAK por los ingenieros de Sun Microsystems™, Java™ fue diseñado para correr en computadoras incrustadas. Sin embargo, en 1995, dada la atención que estaba produciendo la Web, Sun Microsystems™, la distribuyó para sistemas operativos tales como Microsoft® Windows.

El lenguaje mismo se inspira en la sintaxis de C++, pero su funcionamiento es más similar al de Smalltalk que a éste. Incorpora sincronización y manejo de tareas en el lenguaje mismo (similar a Ada) e incorpora interfaces como un mecanismo alternativo a la herencia múltiple de C++.

Java™ llegó a ser extremadamente popular cuando Sun Microsystems™ introdujo la especificación J2EE™ (Java™ 2 Enterprise Edition). Este modelo permite, entre otros, una separación entre la presentación de los datos al usuario (JSP o Applets), el modelo de datos (EJB™), y el control (Servlets). Enterprise Java™ Beans (EJB™) es una tecnología de objetos distribuidos que pudo lograr el sueño de muchas empresas como Microsoft® e IBM® de crear una plataforma de objetos distribuidos con un monitor de transacciones. Con este nuevo estándar, empresas como BEA, IBM®, Sun Microsystems™, Oracle® y otros crearon nuevos *servidores de aplicaciones* que tuvieron gran acogida en el mercado.

Los programas en Java™ generalmente son compilados a un lenguaje intermedio llamado *bytecode*, que luego son interpretados por una máquina virtual (JVM™). Esta última sirve como una plataforma de abstracción entre la máquina y el lenguaje permitiendo que se pueda "escribir el programa una vez, y correrlo en cualquier lado". También existen compiladores nativos de Java, tanto software libre como no libre. El compilador GCC de GNU compila Java™ a código de máquina con algunas limitaciones.

Las especificaciones de la plataforma Java™ las crea y controla el organismo JCP™ (Java™ Community Process) desde 1995. El JCP™ está administrado por el Program Management Office (PMO) compuesto íntegramente por asalariados de SUN™. Los documentos del JCP™ que definen las tecnologías de la plataforma se llaman JSRs y son redactados por grupos de expertos.

Constan de una especificación: un documento que describe la tecnología, su necesidad, y cómo afectará al resto de la plataforma, una implementación de referencia (IR). Demuestra que la tecnología es factible y un test de compatibilidad (TC): batería de pruebas que permiten verificar si una implementación cumple la especificación.
Dentro de los JSR tenemos los UJSR, Umbrella Java™ Specification Request. Son los JSRs que afectan a J2ME™, J2SE™ y J2EE™ y los perfiles de J2ME™. Un miembro de cada grupo que trabaje en un UJSR es siempre un asalariado de SUN™, además SUN™ tiene derecho de veto sobre estas especificaciones.

El hecho de que el JCP™ esté administrado por un grupo, PMO, en el que todos sus miembros son empleados de SUN™ y que además SUN™ tenga derecho de veto sobre ciertas especificaciones hace que no podamos decir que Java™ sea un estándar abierto.

En marzo de 2004 se aprobó un JSR sobre el lenguaje de programación Groovy. Desde que se aprobara este lenguaje la plataforma Java™ pasó a contar oficialmente con un segundo lenguaje de programación, aunque ya existían anteriormente implementaciones de otros lenguajes para la plataforma Java™ entre las que podemos destacar Jython y Jruby.

2.3. Parrot.

Parrot es una máquina virtual diseñada pensando en las necesidades de los lenguajes dinámicamente tipados, como Perl o Python. Parrot deberá de ser capaz de ejecutar programas escritos en esos lenguajes de manera más eficiente que las máquinas virtual diseñadas para lenguajes estáticamente tipados, caso de C# o Java™. Parrot, al igual que CLI, ha sido diseñado para proporcionar interoperatividad entre los lenguajes que son compilados para ella. En teoría, se podrá escribir una clase en Perl, hacer una subclase de esta en Python y luego usar esa subclase en TCL.

Parrot originariamente fue pensada para ser el entorno de ejecución de Perl6, a diferencia de Perl5, el compilador de Perl6 tendrán un compilador y un entorno de ejecución claramente diferenciados.

Actualmente Parrot es capaz de aceptar código ejecutable en cuatro formas:

- PIR (representación intermedia de Parrot), diseñada para ser escrito por personas y compiladores. Oculta los detalles de bajo nivel, como por ejemplo la manera en que los parámetros son pasados a las funciones.

- PASM (ensamblador de Parrot), es un nivel por debajo de PIR, aunque todavía es capaz de ser escrito y leído por humanos y puede ser generada también por compiladores, aunque aquí el autor deberá de tener en cuenta detalles como la llamada a funciones y de registros.

- PAST (árbol abstracto de datos de Parrot) permite a Parrot aceptar una representación de un árbol abstracto de datos muy útil para las personas que escriben compiladores.

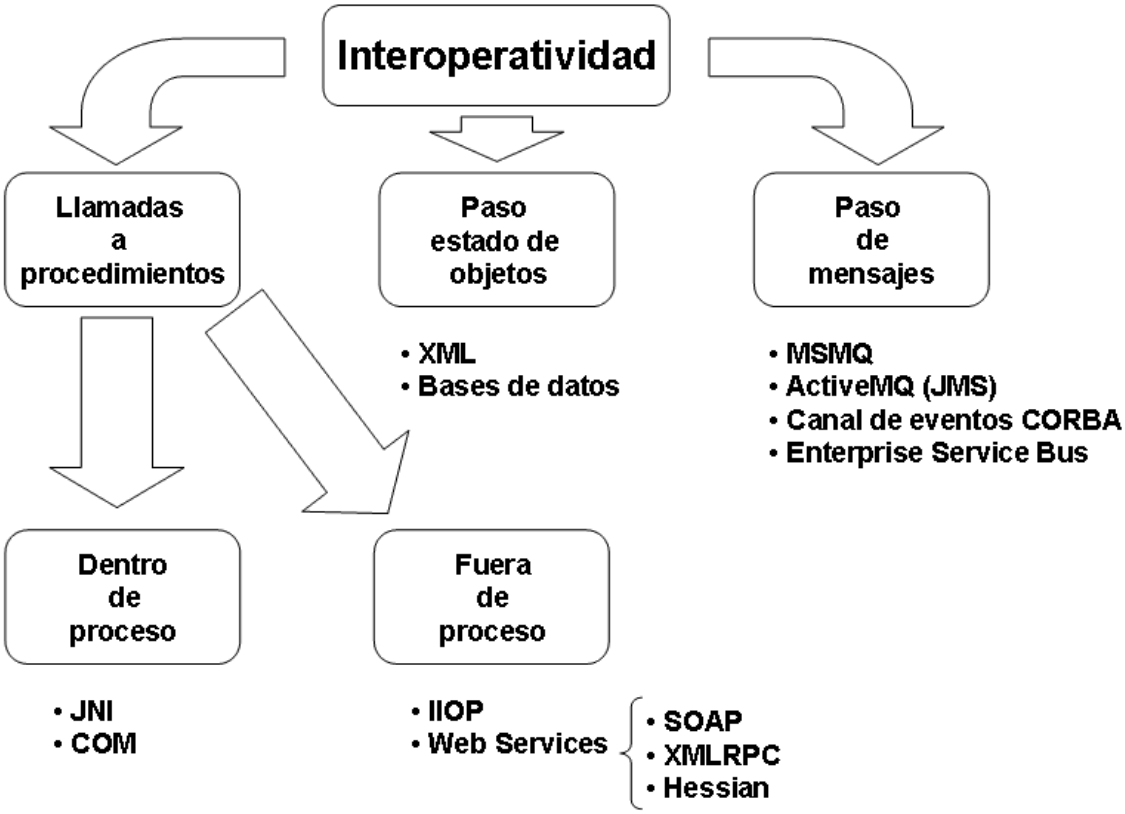
- PBC (*bytecode* de Parrot). Esto es una especie de código máquina para el intérprete de Parrot. No es un lenguaje diseñado para ser interpretado por personas. Todas las anteriores representaciones admitidas por Parrot son automáticamente convertidas dentro de Parrot a PBC. Pero al contrario de las otras representaciones puede ser ejecutada directamente, sin necesidad de una fase de ensamblado. El *bytecode* de Parrot es independiente de plataforma.

Parrot es la única plataforma descrita aquí que no es recomendable para su uso en producción, por ese motivo en el resto del trabajo no se tendrá en cuenta a la hora de estudiar la interoperatividad.

3. Interoperatividad CLI/Java.

En este apartado se tratan varías soluciones de interoperatividad entre las plataformas CLI y Java™.

Las soluciones tratadas aquí se dividen según el tipo de abstracción que proporcionan, estas puede ser basadas en llamadas a procedimiento, paso del estado de objetos o envío de mensajes. Las llamadas a procedimiento pueden ser dentro o fuera del proceso. Las llamadas a procedimientos fuera del proceso se conocen también como llamadas a procedimientos remotos:



Interoperatividad según el tipo de abstracción.

3.1. Interoperatividad a través de COM.

El Component Object Model (COM) es una plataforma de Microsoft® para componentes software publicada en 1993. Se usa para permitir comunicación entre aplicaciones y creación dinámica de objetos es cualquier lenguaje de programación que soporte esa tecnología. El término COM también se usa como denominador común de un conjunto de tecnologías que comprenden además del propio COM, COM+, ActiveX® y DCOM.

A pesar de que existen varías implementaciones es Microsoft® Windows® la plataforma sobre la que se usa normalmente.

3.1.1. Introducción a COM.

Microsoft® ha desarrollado .Net como substituto parcial de COM. Desde el primer momento Microsoft® ha incorporado mecanismos para crear y usar componentes COM desde .Net. El proyecto Mono en sus orígenes fijó entre sus objetivos soportar COM en Windows® y XPCOM, tecnología similar a COM del proyecto Mozilla®, en otras plataformas. Actualmente la única implementación de CLI que soporta COM es la Microsoft®.

COM es una especificación que define a nivel binario como los objetos se deben crear, destruir e interactuar entre ellos. El hecho de que sea un estándar a nivel binario implica que es independiente del lenguaje que se use para trabajar con los objetos, siempre y cuando el lenguaje en cuestión sea capaz de manejar punteros a funciones.

La idea básica en la arquitectura COM es la de tener una serie de componentes, los cuales ofrecen una serie de funcionalidades, y siempre bajo el paradigma de la orientación a objetos. Estos componentes son independientes de cualquier aplicación y residirán en el sistema en forma de DLL's. Cada componente dispone de un identificador globalmente único o GUID (en teoría no debería haber en el mundo dos componentes con el mismo identificador) que lo caracteriza. Si el componente está registrado en el sistema lo podremos *instanciar* a través de nuestras aplicaciones, es decir, crear un objeto concreto de esa clase. Estar registrado en el sistema significa que el sistema conoce donde reside "físicamente" la implementación de la clase identificada por el GUID. En el sistema Windows esta información se almacena en el registro. De esta manera, tenemos a disposición toda una librería de componentes que podemos usar libremente, siempre y cuando estén registrados en nuestro sistema.

Una interfaz un conjunto de métodos (funciones) y atributos (datos) que tienen una fuerte relación lógica entre ellos. En C++, una interfaz es una clase abstracta pura cuyos métodos deben de usar el convenio de llamada *stdcall*, típico del API de Windows®. Un componente es la implementación de una o más interfaces. Una clase queda definida por las interfaces que implementa. Por lo tanto, la herencia en este entorno sólo se concibe a nivel de interfaces, no de implementaciones. De hecho, todas las interfaces derivan del interfaz *IUnknown*. Esta interfaz tiene una funcionalidad vital, ya que se encarga de la gestión de memoria de los objetos. La gestión de memoria se basa en un contador de referencias, y la interfaz *IUnknown* ofrece los métodos *AddRef* y *Release* para controlar este contador. Además, a través del tercer método de esta interfaz, *QueryInterface*, podemos averiguar si el objeto en cuestión implementa una interfaz en particular o no, y obtener un acceso al objeto a través de esa interfaz.

Un componente ActiveX® es un componente COM que se puede inscribir a si mismo en el registro de Windows®. Para que un componente en código nativo, no manipulado según la terminología de .Net, se pueda registrar a si mismo este deberá implementar y exportar las funciones *DllRegisterServer* y *DllUnregisterServer*.

RegSvr32.exe es un programa del sistema operativo que llamará a las funciones *DllRegisterServer* y *DllUnregisterServer* de una determinada .dll nativa para inscribir o borrar los componentes que alberga dicha librería. Si la .dll fuera un ensamblado creado con .Net, Mono y el resto de implementaciones no soportan COM, lo podríamos registrar con la utilidad RegAsm.exe del .Net Framework.

A continuación en los siguientes apartados de este punto vamos a explicar la realización de dos aplicaciones que utilizan COM como intermediario entre .Net y la máquina virtual de Java™ de SUN™.

3.1.2. ActiveX® Bridge.

Como hemos visto en la introducción a COM un componente ActiveX® es un componente COM con la capacidad de revistarse a si mismo.

El ActiveX® Bridge de la máquina virtual de Java™ de Sun™ en Windows permite a un componente basado en la tecnología de componentes de JavaBeans™ actuar como un componente ActiveX, y por consiguiente como componente COM.

El ActiveX® Bridge cuando se usa con el soporte COM de .Net permite usar componentes Java. Esta solución permite si seguimos el modelo de eventos de JavaBeans™ al pie de la letra obtener retro llamadas, con la limitación de que no podremos pasar por parámetros información, desde el componente Java™ en .Net.

Aquí entendemos por retro llamada una invocación de un método en el ejecutable .Net desde el componente Java™.

A continuación vamos a explicar los pasos para desarrollar un componente JavaBean™ que sea capaz de emitir un evento en .Net.

Un JavaBean™ es básicamente una clase pública Java™ con un constructor sin parámetros. Además en el JavaBean™ tendremos que declarar métodos *getPropiedad/setPropiedad* para el acceso y modificación de las propiedades. Para cada evento añadiremos dos métodos *addEvento* y *removeEvento* que nos permiten añadir y quitar receptores. Estos eventos tendrán de parámetro un interfaz que implementamos para los receptores:

```
public class ActiveXBean {

      private CallbackListener callbackListener;

      public ActiveXBean() {
      }
      public String getVirtualMachine() {
            return System.getProperty("java.vm.name");
      }
      public void execCallback(){
            CallbackEvent event = new CallbackEvent(this);
            this.callbackListener.callback(event);
      }
      public void addCallbackListener(CallbackListener
```

```
                                          updateListener) {
            this.callbackListener = updateListener;
      }
      public void removeCallbackListener(CallbackListener
                                          updateListener) {
            this.callbackListener = null;
      }
}
```

Como queremos que el JavaBean™ emita eventos al entorno .Net hemos de crear una clase *BeanInfo* que informe al puente ActiveX® de los eventos que emite el JavaBean™.

El nombre de la clase que informe de los eventos que emite nuestro JavaBean™ será una clase que tenga el mismo nombre que el JavaBean™ más un sufijo *BeanInfo*. Todas las clases del tipo *BeanInfo* deben de implementar interfaz *java.beans.BeanInfo*.

El API de *beans* no da la opción de no tener que implementar todos los métodos de *BeanInfo* extendiendo la clase *java.beans.SimpleBeanInfo*. Como solo necesitamos implementar él método *getEventSetDescriptors* escogemos la segunda opción:

```
public class ActiveXBeanBeanInfo extends SimpleBeanInfo {

   public ActiveXBeanBeanInfo() {
   }
   public EventSetDescriptor[] getEventSetDescriptors() {
   …
   }
}
```

El método *getEventSetDescriptors* devuelve un *array* de elementos *EventSetDescriptor*, un *EventSetDescriptor* indica el conjunto de eventos que lanza un determinado JavaBean™ especificando el nombre del conjunto de eventos, los interfaces que deben implementar los receptores de eventos para recibir el evento y los métodos de esos interfaces que son invocados cuando esto pasa:

```
public EventSetDescriptor[] getEventSetDescriptors() {
      EventSetDescriptor[] arr = new EventSetDescriptor[1];
      Class sourceClass = ActiveXBean.class;
      String eventSetName = "callback";
      Class listenerType = CallbackListener.class;
      String listenerMethodName = "callback";
      try {
            EventSetDescriptor eventSetDescriptor = new
                        EventSetDescriptor(sourceClass,
                        eventSetName, listenerType,
                        listenerMethodName);
            arr[0] = eventSetDescriptor;
            return arr;
      }
      catch (IntrospectionException e) {
            throw new RuntimeException(e.toString());
      }
}
```

Por último escribiremos el interfaz para recepción de eventos, que será el nombre del evento con el sufijo *Listener*:

```
public interface Callback Listener extends
                java.util.EventListener {
          void callback(CallbackEvent callbackEvent);
}
```

El parámetro del método *callback* en el que se reciben los eventos del mismo nombre contiene un parámetro de un tipo derivado de *java.util.EventObject*, se implementa con el nombre del evento terminado en *Event*:

```
public class Callback Event extends java.util.EventObject {
          public CallbackEvent(Object source) {
                super(source);
          }
}
```

Cuando terminemos de programar todo y lo compilemos solamente nos quedará generar un jar con todos los archivos *.class*:

```
jar cvf ActiveXBean.jar activex/*.class
```

Debemos de copiar dicho fichero al directorio *%JAVA_HOME%\axbridge\bin* y ejecutar desde ese directorio el siguiente comando del JDK™:

```
packager.exe ActiveXBean.jar activex.ActiveXBean -reg
```

Este comando crea un envoltorio para el Javabean™ *activex.ActiveXBean* y lo declara como objeto COM en el registro de Windows®.

Podríamos hacer lo mismo de antes en dos pasos:

```
packager.exe ActiveXBean.jar activex.ActiveXBean
regsvr32 ActiveXBean.dll
```

Posteriormente cuando deseemos que el *bean* deje de ser un objeto ActiveX® lo quitaríamos del registro con:

```
regsvr32 /u ActiveXBean.dll
```

Una vez registrado el componente tendremos que crear con el comando tlbimp una .dll manejada que hará de envoltorio de .Net para el componente COM:

```
tlbimp ActiveXBean.dll /out:ActiveXBeanManaged
```

Dentro de el ensamblado generado, si hemos seguido las pasos al pie de la letra, tendremos una clase con el mismo nombre que el JavaBean™ más un sufijo class. Esa clase será la que nos permita usar el JavaBean™ desde .Net. Como hemos seguido el modelo de eventos de Javabeans™ la clase de la dll de .Net también tendrá un evento para recibir las retro llamadas del componente.

```
activeXBean=new ActiveXBeanClass();
activeXBean.callback+=new
ActiveXBeanSource_callbackEventHandler(activeXBean_callback);
activeXBean.execCallback();
```

Los *delegates* que tenemos que crear para recibir los eventos contienen un parámetro *object* que no podremos usar desde .Net como mencionamos antes.

Para pasar información desde Java™ sólo podremos usar métodos que retornen tipos básicos y *strings*:

```
private static void activeXBean_callback(object arg0)
{
      Console.WriteLine(
            activeXBean.getVirtualMachine());
}
```

3.1.3. COM4J.

En la máquina virtual de Sun™ no tenemos ningún método implementado de serie en Windows para acceder a componentes COM.

La máquina virtual de Microsoft® incorporaba la herramienta ActiveX™ Control Importer for Java™ que permitía acceder a componentes COM[1].

En Java™ se encuentran desarrollados varios puentes de comunicación que permiten acceder a COM desde Java, como el comercial J-Integra™ for COM o JACOB bajo licencia LGPL. Para realizar este ejemplo nos hemos decantado por COM4J.

COM4J es un proyecto bajo licencia MIT con dos objetivos:

- Desarrollar una librería que permita a aplicaciones Java™ trabajar con componentes COM.
- Construir una herramienta que importe una librería de tipos COM, archivo TLB, y genere las clases Java™ para poder usarlo a través de la librería del punto anterior.

COM4J está desarrollado para trabajar únicamente con la versión 5.0 de la máquina virtual, además no nos permite realizar ningún tipo de llamada desde COM a Java™ ya que no tiene implementada la manera de implementar interfaces COM en Java.

La versión de COM4J que hemos usado en el ejemplo es la publicada el 30 de Noviembre de 2005.

Un Guid o Identificador Global Único es un número pseudo-aleatorio de 128 bits sirve para identificar componentes e interfaces COM. No se

[1] Las extensiones de la máquina virtual de Microsoft® fueron las causantes de un pleito entre Sun™ y Microsoft® que ha terminado con el acuerdo por parte de Microsoft® de dejar de distribuir y dar soporte el 16 de diciembre de 2005 su propia máquina virtual.

garantiza que todos los Guid sean únicos, aunque la posibilidad de que se repita uno es muy pequeña. Guid es una implementación de Microsoft® de un estándar de la Open Software
Foundation que se llama Identificador Universal Único (UUID).

Para publicar un componente COM se necesitan al menos dos identificadores Guid, uno que identifique al componente COM y otro para designar al interfaz que implementa. Se requiere un Guid para cada interfaz y para cada componente, un componente puede implementar varios interfaces derivados de *IUnknown*, por lo menos uno, y un interfaz puede ser implementado por varios componentes. Generamos los Guid con guidgen.exe, herramienta de Microsoft® para generar identificadores Guid.

El Global Assembly Cache (GAC) es una caché utilizada para almacenar ensamblados que puedan ser utilizados por cualquier aplicación manipulada dentro del equipo local, para publicar un componente COM manipulado es necesario que este esté en el GAC. Para meter un componente en el GAC se requiere que esté firmado. Para firmarlo generamos un par de claves con la utilidad sk:

```
sk -k HelloCOM.snk
```

El componente COM estará contenido en una .dll que deberá contener en alguno de sus fuentes el siguiente meta dato que indique el par de claves con el que se firma:

```
[assembly: AssemblyKeyFile(@"..\..\HelloCOM.snk")]
```

Antes de crear la clase que implemente el componente vamos a escribir el interfaz que va a implementar como componente COM:

```
[ComVisible(true), Guid("41C5AC14-6EDD-422c-81B3-9377299EB366")]
public interface IHelloWorld
{
      void sayHello(string Hello);
      string getVirtualMachine();
}
```

El atributo *ComVisible* con valor *true* indica que en el ensamblado deben de generarse los meta datos para exponer el interfaz como COM, no es necesario ya que por defecto esto se hace. En el atributo *Guid* indicamos el valor del Guid que queremos que tenga el interfaz. Si no pusiésemos este valor el *runtime* le asignaría uno al registrar el componente:

A continuación implementamos el componente mediante una clase que implementará el anterior interfaz:

```
[ComVisible(true), Guid("CFACDF57-E2BA-4f34-BD48-EAF2A336A89B")]
[ProgId("HelloCOM.HelloWorld")]
[ClassInterface(ClassInterfaceType.None)]
public class HelloWorld : IHelloWorld
{
      public void sayHello(string HelloMessage)
      {
            Console.WriteLine(HelloMessage);
      }
```

```
        public string getVirtualMachine()
        {
                return "Microsoft® .NET Runtime Execution Engine";
        }
}
```

Los atributos *ComVisible* y *Guid* significan lo mismo en los interfaces y en los componentes. *ProgId* indica el identificador de programa, por defecto toma el nombre de la clase incluidos los espacios de nombres. Si esta cadena no supera los 39 caracteres a los que está limitado el *ProgId* en COM este atributo no seria necesario. *ClassInterface (ClassInterfaceType.None)* indica al .Net Framework que no me cree ningún interfaz basándose en la clase, como *IHelloWorld* está implementado explícitamente la funcionalidad se expondrá mediante este. Para que el componente pueda usarse con COM4J se requiere este atributo con este valor.

Una vez compilada la dll debemos generar una librería de tipos de COM a partir de esta:

```
tlbexp HelloCOM.dll /out:HelloCOM.tlb
```

Inscribimos los tipos en el registro de Windows:

```
regasm HelloCOM.dll
```

Los dos pasos anteriores los podríamos simplificar con:

```
regasm HelloCOM.dll /tlb:HelloCOM.tlb
```

En el registro de Windows quedará como *IN-PROC Server*, es decir, librería que implementa el componente, mscoree.dll, ya que COM es una tecnología que trabaja a nivel binario y no a nivel de código manipulado. mscoree.dll hace de *proxy* entre COM y nuestro ensamblado, esta librería es la implementación del *runtime* de .Net.

Metemos HelloCOM.dll en el GAC de .Net:

```
gacutil /i HelloCOM.dll
```

Y la importamos con COM4J para que me genere en la carpeta src el paquete Java™ que me servirá para acceder al componente:

```
java -jar tlbimp.jar -o src -p HelloCOM HelloCOM.tlb
```

En el paquete HelloCOM tengo una factoría abstracta con la que crear el componente. El componente se crea con el método estático *createHelloWorld*. En COM siempre se trabaja con interfaces, *createHelloWorld* devuelve una instancia de *IHelloWorld* que será con la que llamamos a los métodos del componente:

```
IHelloWorld iHelloWorld=ClassFactory.createHelloWorld();
iHelloWorld.sayHello("Hello World");
System.out.println(iHelloWorld.getVirtualMachine());
```

Para desinstalar el componente se ejecutarán los siguientes comandos:

Para que deje de ser un objeto COM:

```
regasm /unregister HelloCOM.dll
```

Para que deje de estar HelloCOM en el GAC:

```
gacutil /u HelloCOM
```

3.2. Interoperatividad a través de JNI.

3.2.1. Introducción a JNI.

Existen situaciones en las que no se pueden escribir aplicaciones solamente en Java. JNI es la solución estándar dentro de la tecnología Java™ invocar a código nativo y viceversa.

Además de JNI existen otros interfaces que permiten llamar a procedimientos en código nativo que hoy ya no estar disponibles al ser reemplazados por JNI o eliminados del mercado por acuerdos empresariales, entre estos destacamos:

- JDK™ 1.0 Native Methods Interface (NMI): el JDK™ 1.0 usaba este método, Sun™ reemplazó este método por JNI porque esta solución presentaba dos importantes problemas. Las llamadas a código nativo accedían a estructuras C sin definir la posición exacta que debían ocupar en memoria, lo que causaba que en cada máquina virtual estas llamadas fueran distintas. Cuando se llamaba a un método nativo se creaba un puntero que no se liberaba nunca. El JDK™1.1 usó también esta forma de acceso en las clases incluidas en el *runtime* de Java, aunque ya incorporaba una versión de JNI. Con la versión 1.2 del JDK™ se portaron completamente las clases del *runtime* a JNI.
- Java™ Runtime Inteface (JRI): Fue el método propuesto por Netscape®, tenía la portabilidad en mente. JNI se basó en JRI para su diseño.
- Raw Native Methods (RNI), Java/COM Bridge y JDirect™: Fueron las tecnologías que incorporaron las implementaciones de la máquina virtual de Microsoft®. RNI era muy similar al interfaz de métodos nativos del JDK™, con la particularidad de que para interactuar con el recolector de basura había que usar funciones RNI. RNI no permitía acceder a .dll del API Win32, que comenzó a comercializarse con Windows 95. Por este motivo RNI fue reemplazado por la tecnología JDirect™. La solución a la recolección de basura de JDirect™ era no llamar al recolector de basura cuando se estuviera realizando una llamada al API. Como solución de más alto nivel se podía usar el Java/COM Bridge para llamar a componentes COM. Estas tecnologías dependían de la máquina virtual de Microsoft® que dejó de comercializarse el 16 de diciembre de 2005 por una acuerdo comercial con SUN™.

Existen implementaciones no oficiales de Java™ que no usan una máquina virtual del estilo de la de Java, esas implementaciones aunque implementan JNI incorporan otros métodos para acceso a código máquina distinto del *bytecode* de Java™ como los dos siguientes:

- Compiled Native Inteface: CNI está incluido en el compilador de Gcj, se usa para escribir métodos nativos usando C++ que podrán ser usados por clases Java™ compiladas con Gcj a código nativo. Se basa en usar un subconjunto del ABI C++ en el compilador Gcj para compilar las clases Java. Ofrece mejor desempeño que JNI con código nativo, pero está limitado a este tipo de compilación con Gcj.
- IKVM stubs: IKVM es una máquina virtual que compila Java™ a CIL y ejecuta Java™ con CLI, incorpora versiones para Mono y .Net de JNI. La herramienta ikvmstub.exe de ikvm permite generar archivos jar que hacen de puente entre clases Java™ y clases CIL. El uso de estos jar está limitado a la máquina virtual ikvm.

JNI especifica un API en C y otro en C++ a través de los cuales podemos crear una instaría de la máquina virtual con la que crear instancias de clases Java, asignar valores a sus campos y llamar a sus métodos.

CLI especifica también una manera de invocar métodos nativos a través de la invocación de plataforma (Pinvoke). JuggerNet™ y Caffeine.Net son dos herramientas, la una comercial y la otra bajo licencia MIT, que concatenan el uso de Pinvoke y JNI para ofrecer interoperatividad entre CLI y Java™.

JuggerNet™ permite el acceso de .Net a Java™ y de Java™ a .Net. Además de ser comercial está limitado a .Net. Caffeine.Net es un proyecto con licencia MIT que en el momento de escribir esta memoria no incorporaba la capacidad de acceder a CLI desde Java™. Hemos seleccionado Caffeine.Net como base para hacer un ejemplo de interacción JNI en el siguiente apartado por tratarse de una implementación gratuita, con el código disponible en Internet e incorporar una versión para Mono.

3.2.2. Caffeine.Net.

Caffeine.Net es una solución de interoperatividad entre cualquier máquina virtual que soporte la especificación JNI 1.2 y cualquier entorno de ejecución CLI. Los entornos CLI con los que ha sido testeado Caffeine son el .Net Framework™ de Microsoft® y Mono.

Es normal cometer errores al usar las APIs C y C++ de JNI. Es recomendable que las librerías que hacen de envoltorio JNI para exponer en Java™ funciones o clases escritas en C y en C++ se hagan de manera automatizada, usando por ejemplo SWIG, o siguiendo la norma de que los métodos Java™ guarden una relación de uno a uno con las funciones o métodos de C y C++.

En el caso de Caffeine.Net disponemos de una librería nativa que hace de intermediario entre la máquina virtual y las clases CLI que exponen el API JNI de C. El uso de JNI en C# o cualquier otro lenguaje con soporte de CLI es igual de tedioso que en C. Por este motivo Caffeine.Net también

incorpora generadores de código C# que evitan que tengamos que programar cualquier detalle de JNI.

La versión que vamos a usar de Caffeine.Net es la 0.2.4. . De esta versión solamente está disponible el código fuente.

Para compilar el código fuente en C# podemos usar el Visual Studio®, #develop o cualquier entorno de programación con soporte para un compilador de C#.

Para compilar el código fuente en C necesitamos un compilador de C, no nos vale uno de C++. Nosotros hemos usado el compilador Gcc, en Windows existen varías versiones de gcc, nos decantamos por la versión de MinGW, que genera ejecutables que no requieren de ninguna dll adicional.

El comando para compilar con Gcc en Windows® es:

```
gcc -shared -Os -O3 -DBUILD_DLL -D_JNI_PLATFORM_WIN32
-IC:/Archivos\ de\ programa/Java/JDK™1.5.0_05/include
-IC:/Archivos\ de\ programa/Java/JDK™1.5.0_05/include/win32
-o jninet.dll JVM.c
```

La librería nativa en Windows se llamará jninet.dll y en Linux® libjninet.so.

Para hacer más fácil el proceso de compilación con MinGW existen numerosos entornos de desarrollo. En la realización del ejemplo hemos usado Code::Blocks. Este entorno viene ya con una distribución de MinGW.

Los generadores de código de Caffeine los hemos compilado he introducido en generators.jar.

En Java™ vamos a hacer una clase que llamaremos *HelloWorld* que contendrá dos métodos. Uno para imprimir cadenas de texto por consola y otro para devolver la versión de la máquina virtual como *string*.

```
public class HelloWorld {
      public void SayHello(String HelloMessage){
            System.out.println(HelloMessage);
      }
      public String getVirtualMachine() {
      return System.getProperty("Java.vm.name");
      }
}
```

Compilamos la clase y la introducimos en un jar con el siguiente comando:

```
jar cvf HelloJNI.jar HelloWorld.class
```

Generamos un archivo XML que contenga la defínción del interfaz de esa clase:

```
java -cp generators.jar com.olympum.tools.JavaApiXmlGenerator HelloJNI.jar
HelloJNI.xml
```

Creamos un archivo de fuentes:

```
mkdir src
```

Generamos el archivo envoltorio dentro del directorio de fuentes:

```
java -cp generators.jar com.olympum.tools.CsJniNetWrapperGenerator
HelloJNI.xml src
```

Ahora tenemos una clase *HelloWorld* que hace de *proxy* de la clase que hemos hecho en Java™. El generador de clases tiene un problema, por esto es necesario cambiar dentro de este *proxy* todas las cadenas "java.lang." por una "J". De esta manera en el *proxy* figuran las clases de la librería Caffeine en lugar de los nombres de las clases del entorno de ejecución de Java™.

Para indicarle al entorno de ejecución CLI la máquina virtual Java™ que debe usar y la localización del archivo jar donde debe de buscar las clases hemos de introducir un archivo de configuración App.Config.
El archivo de configuración lo tomamos del ejemplo *Hello* que viene con la distribución de Caffeine.JNI y lo adaptamos a nuestros intereses. Solamente debemos de cambiar dos líneas.

```
<JVM.dll value=
"C:\Archivos de programa\Java\jre1.5.0_05\bin\client\JVM.dll"/>
<Java™.class.path value="HelloJNI.jar"/>
```

Una vez terminados todos estos pasos podremos usar la clase Java™ desde C# de la siguiente manera:

```
HelloWorld helloWorld=new HelloWorld();
helloWorld.SayHello(new JString("Hello World"));
Console.WriteLine(helloWorld.getVirtualMachine().String);
```

Con la máquina virtual de Java™ de Sun™ la salida será la siguiente:

```
Hello World
Java HotSpot(TM) Client VM
```

3.3. Interoperatividad a través de bases de datos.

Una base de datos puede usarse como intermediario entre dos aplicaciones de cualquier tipo que accedan a ella. Tanto para Java™ como para CLI existen librerías estándar con las que usar bases de datos, la librería estándar para Java™ es JDBC™ y su equivalente en CLI es ADO.Net. Estas librerías nos permiten mandar cadenas SQL y recibir información en unos objetos que almacenan el conjunto de resultados que podemos recorrer por medio de cursores.

En este proyecto vamos a usar un enfoque de acceso a bases de datos orientado a objetos para explicar la interoperatividad entre Java™ y CLI a través de una base de datos. Para ello usaremos una técnica conocida como mapeo objeto-relacional.

3.3.1. Introducción al mapeo objeto-relacional.

Mapeo Objeto-SQL, también conocido como mapeo objeto relacional u O/RM, es una técnica de programación consistente en unir bases de datos relacionales a conceptos de lenguajes orientados a objeto, creando así una base de datos orientada a objetos virtual.

En Java, .Net y Mono tenemos una base de datos real orientada a objetos bajo licencia GPL y comercial, db4o. En el caso de no querer pagar la licencia tenemos que recordar que GPL obliga a nuestro programa a usar esa licencia también. El principal problema a la hora de usar este sistema de bases datos para comunicar Java™ y .Net es que la meta información de la base de dato se toma de la plataforma. Las bases de datos db40 son por tanto nativas de la plataforma y dependientes de ella.

En la versión 3.0 de C# el lenguaje incluirá unas extensiones llamadas LINQ[2] con las cuales podremos hacer consultas a bases de datos, documentos XML e incluso dentro de las propias colecciones de objetos de CLI. Con esto ya tendríamos integrado el mapeo Objeto-Relacional dentro del propio lenguaje.

En Java™ existen dos especificaciones sobre persistencia, JDO y EJB™. La persistencia de una estructura o de un objeto consiste en guardar sus características más allá de la ejecución del programa por medio de bases de datos o del sistema de ficheros. Los mapeos objetos relacionales se usan para obtener persistencia.

Los sistemas de bases de datos relacionales guardan una meta información sobre las bases de datos que manejan. Lo natural es partir de una base de datos y crear mediante su meta información las clases y los archivos de mapeo que junto a un entorno de trabajo objeto-relacional comunican el mundo relacional y el de los objetos.

Aunque Hibernate e Ibatis no siguen ningún estándar de Java™ vamos a usarlos para realizar unos sencillos ejemplos sobre mapeo Objeto-Relacional. Hemos escogido estas dos implementaciones por su licencia, LGPL y Apache respectivamente, y por tener implementaciones en Java™ y CLI.

En el fondo para comunicar con la base de datos en CLI y Java™ tanto Hibernate como Ibatis usan ADO.NET y JDBC™. Como base de datos vamos a escoger PostgreSQL 8.1, que tiene un *driver* JDBC™ con licencia BSD y un proveedor ADO.NET, Npgsql, distribuido bajo LGPL. Además PostgreSQL a partir de su versión 8.0 tiene una implementación nativa en Windows®, además de las que ya tenia para Unix® y Linux®. Cuando vamos a escoger un *mapeador* objeto-relacional hemos de fijarnos que soporte la base de datos con la que vamos a trabajar, tanto Hibernate como Ibatis soportan PostgreSQL.

Comenzaremos los ejemplos creando la base de datos y la tabla que vamos a usar. Con PostgreSQL tenemos la utilidad pgAdminIII con la que podremos administrar el servidor de bases de datos desde una interfaz

[2] Lenguaje Integrado de Consultas.

gráfica. Al estar hecha con WxWidgets podremos realizar bases de datos con Linux, FreeBSD y Windows®.

El código SQL con el que generamos la base de datos y la tabla es el siguiente:

```
CREATE DATABASE helloworld;
CREATE TABLE helloworld
(
  id bigserial PRIMARY KEY,
  helloworld varchar(256)
);
```

A partir de aquí ya estamos preparados para trabajar con Hibernate e Ibatis.

3.3.2. Hibernate.

Hibernate es una solución objeto-relacional para el lenguaje Java™. Esta distribuida bajo LGPL.

Hibernate trata de liberar al desarrollador de una cantidad significante de tareas relacionadas con la persistencia. Hibernate genera consultas SQL y realiza la traducción entre el conjunto de resultados y los objetos. Actualmente su principal programador, Gavin King, trabaja para el JBoss™ Group que ofrece soporte al producto.

El mapeo entre los objetos y las tablas de la base de datos se realiza a través de XML y en el caso de usar JDK5.0™ también a través de anotaciones.

NHibernate 1.0 es una versión de Hibernate 2.1 escrita en C#. Comenzó como un proyecto distinto, aunque recientemente se ha unido a Hibernate y al JBoss™ Group.

3.3.2.1. Hibernate con Java™.

Para desarrollar la versión Java™ del ejemplo con Hibernate podremos generar las clases y el xml necesario con el plugin de Eclipse Hibernate Synchronizer.

El código Java™ con Hibernate es muy sencillo, pero para poder trabajar con él tenemos que escribir varios archivos de configuración XML. El archivo de configuración principal es hibernate.cfg.xml y ahí hemos de fijar las siguientes propiedades siguiendo la estructura fijada en el DTD Hibernate Configuration:

- hibernate.connection.url: *url* que usa el *driver* JDBC™. En nuestro caso JDBC:postgresql://localhost:5432/helloworld.
- hibernate.connection.driver_class: clase que implementa el *driver* JDBC™ , org.postgresql.Driver para PostgreSQL
- hibernate.connection.username: nombre de usuario de la base de datos.
- hibernate.connection.password: password de usuario.

- dialect: el dialecto fija la manera en que se genera el SQL, hay uno para cada base de datos soportada por Hibernate. El valor para PosgreSQL es *org.hibernate.dialect.PostgreSQLDialect*
- hibernate.show_sql: útil para depurar los programas, muestra el SQL generado por consola.

Además de fijar propiedades le informamos a Hibernate de los archivos de mapeo que vamos a utilizar:

```
<mapping resource="Helloworld.hbm" />
```

Helloworld.hbm es un archivo XML que sigue el DTD Hibernate-mapping. Aquí hemos de especificar el nombre de la tabla en la base de datos y la clase Java™ usada en la persistencia. Para cada relación tabla clase tendremos que detallar que relación tiene cada una de las propiedades de la clase Java™ con cada columna de la tabla y el tipo JDBC™ usado para recuperar y guardar la información. También hemos de definir la relación entre la clave de la tabla y la propiedad especial que Hibernate obliga que exista como identidad del objeto.

Para la identidad del objeto además tenemos que concretar como se genera. Los tipos *serial* y *bigserial* de PostgreSQL llevan asociada una secuencia que es la usada para generar los valores identidad:

```
<hibernate-mapping package="helloworld">
<class name="Helloworld" table="helloworld">
      <id column="id" name="Id" type="long">
      <generator class="sequence">
            <param name="sequence">helloworld_id_seq</param>
      </generator>
      </id>
      <property column="helloworld" length="255 name="Helloworld"
                              not-null="false" type="string"/>
</class>
</hibernate-mapping>
```

En el archivo de mapeo especificamos que existe una clase Java™ llamada *Helloworld*. Esta clase debe de tener un constructor sin argumentos y las propiedades descritas en el archivo de mapeo:

```
public class Helloworld implements Serializable {
      private String _helloworld;
      private long _id;

      public Helloworld() {}

      public java.lang.String getHelloworld() {
            return _helloworld;}
      public void setHelloworld(java.lang.String _helloworld) {
            this._helloworld = _helloworld;}
      public long getId() { return _id; }
      public void setId(long _id) { this._id = _id; }
}
```

A partir de aquí ya tenemos toda la infraestructura requerida por Hibernate. A continuación vamos a usarlo.

Tenemos que meter en nuestro *classpath* los jar hibernate3.jar, commons-collection-2.1.1.jar, commons-loggin-1.0.4.jar, dom4j-1.6.jar, ehcache-1.1.jar, jta.jar, cglib-2.1.jar, asm.jar, antlr-2.7.5H3.jar y el *driver* JDBC™ postgresql-8.1dev-403.JDBC3.jar.

Hibernate usa Commons-logging para emitir mensajes de log. Nosotros no queremos que se emitan mensajes que no sean errores. Para ello vamos a añadir el siguiente bloque *static* a nuestra clase principal, en el añadimos dos propiedades para que las interprete commons-logging. Fijamos *org.apache.commons.logging.Log* para indicarle que use la implementación de logging *org.apache.commons.logging.impl.SimpleLog*. Esta implementación saca los mensajes por pantalla. Si no se fija esta variable commons-loggin intentará emitir los mensajes de log a través de log4J, posteriormente emitirá un error porque no hemos incluido log4J en el *classpath* y continuará usando SimpleLog. El objetivo de fijar *org.apache.commons.logging.Log* es quitar este mensaje de error. Posteriormente también le daremos valor a *org.apache.commons.logging.simplelog.defaultlog*, el valor *Error* le indica a *SimpleLog* que no emita mensajes que no sean errores. La implementación es:

```
static {
      System.setProperty("org.apache.commons.logging.Log",
                  "org.apache.commons.logging.impl.SimpleLog");
      System.setProperty(
            "org.apache.commons.logging.simplelog.defaultlog",
                     "error");
}
```

Para trabajar con Hibernate tenemos que leer el archivo hibernate.cfg.xml, lo hacemos con:

```
Configuration cfg = new Configuration();
cfg.configure();
```

Con la configuración vamos a abrir una sesión:

```
SessionFactory sf=cfg.buildSessionFactory();
Session session=sf.openSession();
```

Las sesiones deben de cerrarse:

```
session.close();
```

Cuando queramos agrupar las modificaciones en la base de datos Hibernate soporta transacciones. Las transacciones se implementan creando un objeto transacción con el método *beginTransaction* del objeto sesión, para ejecutar la transacción usaremos el método *commit* de *transaction*:

```
Transaction transaction=session.beginTransaction();
...
transaction.commit();
```

Si queremos crear un objeto en la base de datos lo creamos en Java™ y llamamos al método *save* de *session* que nos devolverá la identidad de ese objeto:

```
long id=((Long)session.save(helloWorld)).longValue();
```

Para actualizar un objeto usaremos *update*:

```
session.update(helloWorld);
```

Si queremos recuperar un único objeto de la base de datos con una determinada identidad lo haremos con *load*:

```
helloWorld=(Helloworld)session.load(Helloworld.class,
                                    new Integer(id));
```

En el caso de querer recuperar varios objetos de la base de datos haremos una consulta en HQL[3] con el método *createQuery* que nos devuelve un objeto *Query*, para efectuar la consulta usaremos el método *list* de ese *Query*:

```
List consulta=session.createQuery("from Helloworld").list();
```

3.3.2.2. NHibernate.

Para desarrollar la versión CLI podremos generar las clases y el XML necesario con la herramienta MyGeneration. MyGeneration es una herramienta escrita en C# que combina los meta datos de las bases de datos con un procesador de *scripts* para generar entre otras cosas procedimientos almacenados, objetos persistentes e interfaces gráficos.

El código C# con NHibernate es muy corto, pero al igual que Hibernate con Java™ requiere de varios archivos de configuración. En el App.config de la aplicación hemos de configurar los siguientes parámetros:

- hibernate.connection.provider: El proveedor ADO.Net, con PostgreSQL es *NHibernate.Connection.DriverConnectionProvider*.
- hibernate.dialect: el dialecto fija la manera en que se genera el SQL, hay uno por cada base de datos soportada por NHibernate. El valor para PosgreSQL es *NHibernate.Dialect.PostgreSQLDialect*.
- hibernate.connection.driver_class: clase que implementa el *driver* ADO.Net, *NHibernate.Driver.NpgsqlDriver* para PostgreSQL.
- hibernate.connection.connection_string: *string* que le pasamos a ADO.Net para conectarnos con la base de datos. En nuestro caso le hemos dado el valor siguiente : "Server=127.0.0.1;Port=5432;User Id=postgres;Password=postgres;Database=helloworld;Encoding=UNICODE"
- hibernate.show_sql: útil para depurar los programas, muestra el SQL generado por consola.

Hellworld.hbm.xml es un archivo XML que sigue el DTD Hibernate-mapping. Lo empotraremos como recurso incrustado. Aquí hemos de definir el nombre de la tabla en la base de datos y la clase CLI usada en la persistencia. Para cada relación tabla clase tendremos que especificar que relación tiene cada una de las propiedades de la clase Java™ con cada columna de la tabla y el tipo CLI usado para recuperar y guardar la

[3] Lenguaje de consultas de Hibernate

información. También hemos de detallar la relación entre la clave de la tabla y la propiedad especial que Hibernate obliga que exista como identidad del objeto. Para la identidad del objeto además tenemos que explicar como se genera. Los tipos *serial* y *bigserial* de PostgreSQL llevan asociada una secuencia que es la usada para generar los valores identidad:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
<class name="HelloNHibernate.HelloWorld,HelloNHibernate"
                                        table="helloworld">
      <id name="id" column="id" type="Int64" unsaved-value="0">
            <generator class="sequence">
            <param name="sequence">helloworld_id_seq</param>
            </generator>
      </id>
      <property column="helloworld" name="helloworld"
                                              type="String" />
</class>
</hibernate-mapping>
```

En el archivo de mapeo especificamos que existe una clase CLI llamada *Helloworld*. Esta clase debe de tener un constructor sin argumentos y las propiedades descritas en el archivo de mapeo:

```
public sealed class HelloWorld
{
      private Int64 m_id;
      private String m_helloworldname;

      public HelloWorld(){}

      public Int64 id
      {
            get { return m_id; }
            set { m_id = value; }
      }



      public String helloworld
      {
            get { return m_helloworldname; }
            set { m_helloworldname = value; }
      }
}
```

Debemos de referenciar en el proyecto los dlls Castle.DynamicProxy.dll, HashCodeProvider.dll, Iesi.Collections.dll,log4net.dll, Mono.Security.dll, NHibernate.dll y como *driver* Ado.Net a Npgsql.dll.

Con todo esto ya tenemos toda la infraestructura requerida por Hibernate, vamos a usarlo.

Para trabajar con NHibernate tenemos que leer el archivo App.Config, lo hacemos con:

```
Configuration cfg = new Configuration();
```

A diferencia de Hibernate aquí necesitamos además de la configuración los archivos de mapeo introducidos como recursos incrustados:

```
cfg.AddAssembly(Assembly.GetExecutingAssembly());
```

Con la configuración vamos a abrir una sesión:

```
ISessionFactory factory = cfg.BuildSessionFactory();
ISession session = factory.OpenSession();
```

Las sesiones deben de cerrarse:

```
session.Close();
```

Cuando queramos agrupar las modificaciones en la base de datos NHibernate soporta transacciones. Las transacciones se implementan creando un objeto transacción con el método *beginTransaction* del objeto *session*, para ejecutar la transacción usaremos el método *commit* de *transaction*:

```
ITransaction transaction = session.BeginTransaction();
...
transaction.Commit();
```

Si queremos crear un objeto en la base de datos lo creamos en Java™ y llamamos al método *Save* de *session* que nos dará y nos devolverá la identidad de ese objeto:

```
long id=(long)session.Save(helloWorld)
```

Para actualizar un objeto usaremos *Update*:

```
session.Update(helloWorld);
```

Si queremos recuperar un único objeto de la base de datos con una determinada identidad lo haremos con *Load*:

```
helloWorld=(HelloWorld)session.Load(typeof(HelloWorld)
                                              ,(long)Id);
```

En el caso de querer recuperar varios objetos de la base de datos haremos una consulta en HQL con el método *Find*. Nos devuelve una lista con la respuesta:

```
IList query=session.Find("from HelloWorld");
```

3.3.3. Ibatis.

Ibatis es una herramienta para ayudar a implementar capas de persistencia en Java™ y en CLI.

Se compone de dos entornos de trabajo: SQLMaps es una solución para el problema de la persistencia que permite *mapear* tablas SQL y objetos. No es una herramienta objeto-relacional completa como Hibernate. DAO es otro entorno de trabajo cuyo objetivo es abstraer los detalles de la

implementación de la persistencia, como su nombre indica está basado en el patrón DAO y puede usarse por encima de SQLMaps. Actualmente Ibatis se encuentra bajo el paraguas del proyecto Apache.

Nosotros vamos a usar para los ejemplos SQLMaps con las implementaciones de Ibatis para Java™ y CLI.

3.3.3.1. Ibatis SQLMaps Java™.

La versión de Ibatis SQLMaps en Java™ que vamos a usar es la 2.1.5, es la equivalente a la versión 1.2.1 de CLI.

Ibatis es mucho más sencillo que Hibernate, solamente requiere el jar del *driver* JDBC™, en nuestro caso postgresql-8.1dev-403.JDBC3.jar, el jar de SQLMaps ibatis-sqlmap-2.jar y las clase comunes de Ibatis ibatis-common-2.jar.

Comenzamos haciendo un archivo de configuración siguiendo el DTD sql-map-config-2.dtd. En el fijaremos las siguientes propiedades:

- JDBC.ConnectionURL : url que usa el driver JDBC™. En nuestro caso JDBC:postgresql://localhost:5432/helloworld.
- JDBC.Driver: clase que implementa el *driver* JDBC™ , *org.postgresql.Driver* para PostgreSQL.
- JDBC.Username: nombre de usuario de la base de datos.
- JDBC.Password: clave de usuario.

En este archivo también le indicamos la dirección del archivo de mapeo que vamos a usar, helloworld_SqlMap.xml en nuestro caso.

En los archivos de mapeos siguiendo la estructura del DTD sql-map-2.dtd, para cada relación tabla clase tendremos que especificar que relación tiene cada una de las propiedades de la clase Java™ con cada columna de la tabla y el tipo JDBC™ usado para recuperar y guardar la información:

```
 <resultMap id="HelloworldResult" class="Helloworld">
      <result column="id" property="id" JDBCType="BIGINT" />
      <result column="helloworld" property="helloworld"
      JDBC™ Type="VARCHAR" />
</resultMap>
```

Como Ibatis no implementa dialectos[4] como Hibernate para escribir SQL tendremos que añadir nosotros *templates* SQL con los que realizar las consultas. Ibatis mezclará los *templates* con los valores de los objetos.

Así tenemos *templates* para insertar que devuelven la clave del objeto insertado:

[4] Los dialectos son las herramientas que utiliza Hibernate para generar SQL para una determinada base de datos.

```
<insert id="insert" parameterClass="Helloworld">
 insert into public.helloworld (helloworld)
 values (#helloworld:VARCHAR#)
 <selectKey resultClass="java.lang.Long" keyProperty="id">
   select CURRVAL('helloworld_id_seq')
 </selectKey>
</insert>
```

Templates para modificar tablas:

```
<update id="update" parameterClass="Helloworld">
     UPDATE USER
     SET helloworld = #helloworld#
     WHERE id = #id#
</update>
```

Templates para borrar registros:

```
<delete id="deleteUser"
       parameterClass="java.lang.Long">
    DELETE
    FROM  USER
    WHERE ID = #VALUE#
</delete>
```

Y por supuesto para hacer consultas:

```
<select id="selectByPrimaryKey" resultMap="HelloworldResult"
    parameterClass="java.lang.Long">
    select id,
          helloworld
    from helloworld
    where id = #VALUE#
</select>
```

Para trabajar con Ibatis tenemos que leer el archivo helloIbatis.xml, lo hacemos con:

```
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient
           (Resources.getResourceAsReader("helloIbatis.xml"));
```

Si queremos crear un objeto en la base de datos lo creamos en Java™ y llamamos al método *insert* que creará un registro en la base de datos y nos devolverá la identidad de ese objeto:

```
long id=((Long)sqlMap.insert("insert",helloworld))).longValue();
```

Para actualizar un objeto usaremos *update*:

```
sqlMap.update("update",helloworld);
```

Si queremos recuperar un único objeto de la base de datos con una determinada identidad lo haremos con *queryForObject*:

```
Helloworld helloworld= (Helloworld)
      sqlMap.queryForObject("selectByPrimaryKey",new Long(id));
```

Para obtener una lista de elementos desde una consulta usaremos el método *queryForList*:

```
List list=sqlMap.queryForList("selectAll",null);
```

3.3.3.2. Ibatis SQLMaps CLI.

La versión de Ibatis SQLMaps .Net que vamos a usar es la 1.2.1, es la equivalente a la versión 2.1.5 de Java™.

Ibatis es mucho más sencillo que NHibernate, solamente requiere las .dll del proveedor ADO.Net, en nuestro Mono.Security.dll y Npgsql, la dll de SQLMaps IBatisNet.DataMapper.dll, las clases comunes de Ibatis IBatisNet.Common.dll, para generar *proxys* Castle.DynamicProxy.dll y para los *loggers* log4net.dll.

En la distribución de Ibatis tenemos un archivo XML con descripciones de como se deben de usar los proveedores con el entorno de trabajo. Tendremos que copiar ese archivo al directorio que contiene el ejecutable de nuestra aplicación y dar el valor *true* al parámetro *enabled* de la etiqueta XML donde se describe el proveedor que vamos a usar, con PostgreSQL usaremos PostgreSql0.7.

Debemos hacer un archivo de configuración SQLMap.config siguiendo el esquema SqlMapConfig.xsd en el que escribiremos la cadena de conexión con la base de datos y el proveedor a usar:

```
<database>
      <provider name="PostgreSql0.7" />
      <dataSource name="helloworld"
connectionString="Server=127.0.0.1;Port=5432;User Id=postgres;
                  Password=postgres;Database=helloworld;" />
</database>
```

En este archivo también le indicamos la dirección del archivo de mapeo que vamos a usar, helloworld.xml en nuestro caso:

```
<sqlMaps>
      <sqlMap resource="Helloworld.xml" />
</sqlMaps>
```

En los archivos de mapeos siguiendo la estructura del *schema* SqlMap.xsd, para cada relación tabla clase tendremos que especificar que relación tiene cada una de las propiedades de la clase Java™ con cada columna de la tabla y el tipo JDBC™ usado para recuperar y guardar la información:

```
<resultMaps>
      <resultMap id="HelloworldResult" class="Helloworld">
      <result property="HelloworldName" column="helloworld"
            type="string" dbType="varchar"/><result property="Id"
            column="id" type="long" dbType="long"/></resultMap>
</resultMaps>
```

Como Ibatis no implementa dialectos como Hibernate para escribir SQL tendremos que añadir nosotros *templates* SQL con los que realizar las consultas. Ibatis mezclará los *templates* con los valores de los objetos.

Así tenemos *templates* para insertar que devuelven la clave del objeto insertado:

```
<insert id="insert" parameterClass="Helloworld">
      insert into public.helloworld (helloworld)
      values (#helloworld:VARCHAR#)
      <selectKey resultClass="long" keyProperty="id">
                  select CURRVAL('helloworld_id_seq')
      </selectKey>
</insert>
```

Templates para modificar tablas:

```
   <update id="update" parameterClass="Helloworld">
        UPDATE USER
        SET helloworld = #helloworld#
        WHERE id = #id#
  </update>
```

Templates para borrar registros:

```
  <delete id="deleteUser"
         parameterClass="long">
      DELETE
      FROM  USER
      WHERE ID = #VALUE#
  </delete>
```

Y por supuesto para hacer consultas:

```
  <select id="selectByPrimaryKey" resultMap="HelloworldResult"
      parameterClass="long">
      select id,
            helloworld
      from helloworld
      where id = #VALUE#
  </select>
```

Para trabajar con Ibatis tenemos que leer el archivo sqlMap.config, lo hacemos con:

```
SqlMapper sqlMapper=SqlMapper.Configure();
```

Si queremos crear un objeto en la base de datos lo creamos en C# o cualquier otro lenguaje que soporte CLI y llamamos al método *Insert* que insertará un registro y nos devolverá la identidad de ese objeto:

```
long id=(long)sqlMappper.Insert("insert",helloworld);
```

Para actualizar un objeto usaremos *Update*:

```
sqlMapper.Update("update",helloworld);
```

Si queremos recuperar un único objeto de la base de datos con una determinada identidad lo haremos con *queryForObject*:

```
Helloworld helloworld= (Helloworld)
      sqlMapper.QueryForObject("selectByPrimaryKey",id);
```

Para obtener una lista de elementos desde una consulta usaremos el método *queryForList*:

```
IList list=sqlMapper.QueryForList("selectAll",null);
```

3.4. Interoperatividad a través de documentos XML.

XML es un lenguaje de marcas del W3C de propósito general. Es una simplificación del SGML. Su principal propósito es facilitar compartir información entre sistemas heterogéneos.

En el siguiente apartado vamos a describir los pasos necesarios para crear dos objetos persistentes en CLI y Java™ respectivamente con capacidad para guardar su estado en documentos XML que comparten la misma estructura definida con un Schema.

3.4.1. Introducción a XML, XML Schema y persistencia.

XML Schema, también llamado "Esquema", es un documento de definición estructural al estilo de los DTD, que además cumple con el estándar XML. Los documentos Schema (usualmente con extensión XSD) se concibieron como un sustituto de los DTD teniendo en cuenta los puntos débiles de estos y la búsqueda de mayores y mejores capacidades a la hora de definir estructuras para los documentos XML, como la declaración de los tipos de datos.

La persistencia de una estructura o de un objeto consiste en guardar sus características más allá de la ejecución del programa por medio de bases de datos o del sistema de ficheros. Un objeto que no sea persistente vive sólo durante la ejecución del programa y su información se pierde al terminar el programa.

La solución que vamos a plantear para la interoperabilidad entre Java™ y CLI con XML se basará en XMLBeans y *serialización* de .Net usando los *schemas* generados con xsd.exe como entrada para el compilador de *schemas* de XMLBeans scomp.

Xmlbeans es una herramienta para acceder a documentos XML vinculándolos a tipos Java™. Se basa en usar las características de XML y XML Schema para *mapear* un tipo de documentos XML de la forma más sencilla posible en clases y tipos Java™. XMLBeans usa XML Schemas para generar interfaces Java™ y clases con las que acceder y modificar instancias XML.

El .Net Framework SDK y mono poseen una herramienta con las que podemos crear XML-Schemas a partir de ensamblados, además del paso contrarío, generar clases con XML-Schemas. Esta herramienta es xsd.exe. Mediante la *serialización* XML de CLI tendremos las posibilidad de, al igual que con XMLBeans, acceder y modificar instancias XML.

3.4.2. *Serializador* XML de CLI.

Como XMLBeans no genera *schemas* a partir de código Java™ vamos a comenzar con CLI. El primer paso es crear una clase al estilo de los *beans*, con un constructor sin parámetros:

```
[XmlRoot(Namespace="HelloWorld")]
public class HelloWorld
{
      private string _helloWorldname;
      private long _id;
      public HelloWorld()
      {
            _helloWorldname = null;
            _id = 0;
      }
      [XmlElement(ElementName = "HelloWorld")]
      public string HelloWorldName
      {
            get { return _helloWorldname; }
            set { _helloWorldname = value; }
      }
      public long Id
      {
            get { return _id; }
            set { _id = value; }
      }
}
```

Usamos atributos para personalizar el *schema* que generaremos con xsd.exe. *XmlRoot* nos permitirá indicarle el atributo *targetNamespace* de la etiqueta *schema* que XmlBeans usará como nombre del paquete que contendrá a las clases que generaremos. En C# ningún componente excepto los constructores puede llamarse igual que la clase que los contiene. Nosotros queremos que en el *schema* el *string* y el tipo complejo, la estructura *helloworld*, se llamen igual. Esto lo arreglamos con el atributo *XmlElement*.

La clase que nos permite guardar y recuperar el contenido de un objeto en XML es *xmlSerializer*. Para crear un objeto desde XML haríamos:

```
XmlSerializer xmlSerializer = new
                              XmlSerializer(typeof (HelloWorld));
StreamReader streamReader = new StreamReader("helloworld.xml");
helloWorld = (HelloWorld)
                        xmlSerializer.Deserialize(streamReader);
StreamReader.Close();
```

Para guardar el estado de un objeto en XML:

```
XmlSerializer xmlSerializer =
                        new XmlSerializer(typeof (HelloWorld));
StreamWriter streamWriter = new StreamWriter("helloworld.xml");
xmlSerializer.Serialize(streamWriter, helloWorld);
streamWriter.Close();
```

Para obtener el XML Schema que siguen los documentos XML que leemos o generamos usaremos xsd.exe, herramienta contenida en el .Net Framework SDK y en Mono.

```
xsd HelloXMLBeans.exe /t:HelloWorld
move schema0.xsd HelloWorld.xsd
```

El archivo HelloWorld.xsd será el que usemos como entrada del ejemplo con XmlBeans.

3.4.3. XmlBeans.

La versión de XMLBeans que vamos a usar es la 2.1.

Con XmlBeans hemos de partir de un XML schema, el generado en el apartado anterior, con el que crearemos usando el compilador scomp, suministrado con el software, un jar con las clases que necesitamos para leer y escribir XML:

```
scomp HelloWorld.xsd  -out helloWorld.jar
```

Deberemos incorporar al *classpath* de nuestra aplicación en Java™ el helloWorld.jar generado junto a los jar jsr173_1.0_api.jar y xbean.jar que obtendremos de la distribución XmlBeans.

El jar generado contiene un paquete Java™ que se corresponde con el namespace que indicamos mediante el attributo *XmlRoot* en .Net. En ese paquete tendremos los interfaces *helloWorldDocument* y *HelloWorld* con los que haremos la *serialización* y representaremos el elemento compuesto de XML *HelloWorld*.

Para leer el objeto desde XML los pasos que debemos hacer son:

```
helloWorldDocument=HelloWorldDocument.Factory.parse(
                                new File("helloworld.xml"));
helloWorld=helloWorldDocument.getHelloWorld();
```

Si lo que queremos es crear un nuevo objeto:

```
helloWorldDocument=HelloWorldDocument.Factory.newInstance();
helloWorld=helloWorldDocument.addNewHelloWorld();
```

Por último para guardarlo:

```
helloWorldDocument.save(new File("helloworld.xml"));
```

3.5. Interoperatividad con IIOP.

IIOP™ es un protocolo de Internet especificado en el estándar industrial CORBA™.

Common Object Request Broker Architecture (CORBA™) es un estándar que establece una plataforma de desarrollo de sistemas

distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

3.5.1. Introducción a IIOP.

CORBA™ fue definido y está controlado por el Object Management Group (OMG) que define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida.

Un ORB (Object Request Broker) es un *middleware* que permite hacer y tratar peticiones transparentemente sobre objetos localizados local o remotamente. CORBA™ define una especificación de ORBs y un protocolo general para la comunicación entre estos ORBs que denomina GIOP. GIOP especifica una sintaxis de transferencia estándar y un conjunto de formatos de mensajes para comunicaciones entre ORBs, aunque no especifica los medios a través los mensajes han de ser transmitidos. IIOP™, protocolo Inter-ORBs de Internet, concreta el protocolo GIOP añadiendo la manera de utilizar conexiones TCP/IP para transmitir mensajes. IIOP™ es el único protocolo que obligatoriamente debe estar en los ORBs CORBA™ 2.0.; aunque existen más protocolos basados en GIOP que definen otras maneras de transmitir mensajes como DIOP, a través de UDP y solamente en un sentido; MIOP, sobre multicast/IP o XIOP, usando XML1.0 y HTTP1.1.

IIOP™ fue diseñado para comunicar ORBs que siguieran la normativa CORBA™ y este es el cometido de la mayoría de las implementaciones de este protocolo; pero también podemos encontrar otras implementaciones de IIOP™ que no forman parte de ORBs CORBA™ y que pueden ínter operar con estos al usar el mismo protocolo.

RMI es un mecanismo para hacer programas distribuidos en Java™. Permite que un objeto que reside en una máquina virtual de Java™ pueda hacer peticiones a otro situado en otra máquina virtual. Como protocolo de comunicación de RMI se desarrolló JRMP, Java™ Remote Method Protocol. JRMP fue pensado para usarse solamente entre máquinas virtuales Java, por lo que los desarrolladores que querían comunicar sus programas con otros de otras plataformas se veían obligados a usar otros mecanismos tales como los citados ORBs de CORBA™.

Para facilitar la labor de los programadores y permitirles desarrollar aplicaciones sin tener que aprender otro lenguaje que no sea Java™ con un estándar de comunicación no propietario se desarrolla RMI/IIOP. RMI/IIOP permite usar el protocolo IIOP™ con Java™ sin tener que conocer el lenguaje IDL, lenguaje con el que se definen los interfaces en CORBA™ y que es imprescindible conocer para desarrollar con ORBs de CORBA™. RMI/IIOP presenta una ventaja en lo que respecta a la interoperatividad frente al RMI clásico con JRMP, RMI/IIOP puede comunicarse con cualquier implementación de CORBA™ 2.0.

Cuando queremos que un programa escrito con RMI/IIOP interactúe con un ORBs CORBA™ necesitaremos generar el código IDL, pero la herramienta rmic del *framework* RMI lo escribirá por nosotros partiendo de

los interfaces RMI en Java™. La manera en que rmic genera el IDL también se encuentra estandarizada dentro de CORBA™.

RMI/IIOP a diferencia de RMI/JRMP no incluye ningún recolector de basuras distribuido basado en tiempos, por este motivo deberemos de especificar con el método *PortableRemoteObject.unexportObject* cuando queramos que un objeto deje de estar disponible por red. RMI/IIOP lo encontramos en el entorno de ejecución estándar de SUN™ desde la versión 1.3.

Net Remoting es un mecanismo de comunicación implementado en el .Net Framework y en Mono. Un canal de Remoting es un objeto que transporta mensajes entre dos aplicaciones. Remoting incorpora de serie tres canales Remoting que son TcpChannel, sobre TCP/IP; HttpChannel sobre HTTP e IpcChannel, incluido a partir de la versión 2.0 de .Net Framework y que usa Pipes con nombre de Windows®. Además de estos tenemos en el mercado otros canales que implementan distintos mecanismos de comunicación para o bien conseguir mejor rendimiento o interoperatividad con otras plataformas.

Disponemos de dos canales Remoting que implementan IIOP™ que son Remoting.CORBA™ e IIOP.Net. La implementación IIOP.Net además del canal incorporar herramientas para generar IDL desde código intermedio y viceversa. Por este motivo esta es la que hemos seleccionando para el ejemplo del siguiente apartado.

Debemos de recordar que IIOP.Net es un canal Remoting y no una implementación completa de un ORB. Por este motivo hay que tener en cuenta que Remoting usa un mecanismo de tiempos para calcular el tiempo de vida de los objetos distribuidos. Este tiempo como veremos en el ejemplo del siguiente apartado puede fijarse como infinito. En Remoting disponemos del método *RemotingServices.Disconnect*, análogo al método *PortableRemoteObject.unexportObject* de RMI/IIOP, para indicarle que ese objeto deja de estar disponible a través de red.

3.5.2. RMI/IIOP e IIOP.Net.

En el siguiente ejemplo vamos a usar la versión 1.7.1 de IIOP.Net el JDK5.0™ de SUN™.

Comenzaremos haciendo un cliente y un servidor con RMI/IIOP con una retro llamada o *callback*. Posteriormente a partir de RMI/IIOP generaremos IDL y haremos a partir de ahí un servidor y un cliente en C#. Con este procedimiento curiosamente el servidor y el cliente escritos en C# con IIOP.Net no son capaces de realizar el *callback* entre ellos, aunque funcionan perfectamente cuando trabajan con el servidor y el cliente RMI/IIOP.

Una retro llamada o *callback* en CORBA™ es un proceso a través del cual el servidor puede tomar la iniciativa de iniciar una comunicación, es decir, llamar a un método de un objeto hospedado en el cliente. El cliente publica un objeto y le pasa la referencia del mismo al servidor a través de un parámetro en la llamada a un método con una referencia de un objeto

contenido en el servidor. Cuando el servidor posee una referencia a un objeto publicado por un cliente estos pueden invertir sus papeles. Una retro llamada no es más que una llamada a un método con una referencia que se ha obtenido por este proceso.

En todas las comunicaciones tendremos que tener arrancado un servidor de nombres CORBA™, en el caso de Java™ se arranca con el siguiente comando:

```
orbd -ORBInitialPort 3000
```

3.5.2.1. Servidor RMI/IIOP.

Comenzaremos escribiendo los interfaces como en cualquier aplicación RMI. Escribiremos dos, uno para el servidor y otro para el *callback*. Todos los métodos de los interfaces RMI deben de ser etiquetados para lanza la excepción *RemoteException*:

```
public interface HelloWorld extends Remote {
      public String getVirtualMachine() throws RemoteException;
      public void execCallback(HelloWorldCallback
                  helloWorldCallback) throws RemoteException;
      public void sayHello(String hello) throws RemoteException;
}

public interface HelloWorldCallback extends Remote {
      public void callback(String message)
                              throws RemoteException;
}
```

El servidor se implementará en una clase que derive de *javax.rmi.PortableRemoteObject* e implemente *Helloworld*. Aquí encontramos una diferencia con RMI, donde las clases que hacen de servidor derivan de *java.rmi.server.UnicastRemoteObject*.

Cuando ya tengamos esta clase y los interfaces compilados con rmic vamos a generar los cabos del cliente y los esqueletos del servidor necesarios para trabajar con RMI/IIOP:

```
rmic -iiop -idl HelloWorldImpl
```

En este paso también generaremos el IDL con el que posteriormente trabajaremos con IIOP.Net. En este paso se creará también el IDL del *callback* ya que es necesario incluirlo en el IDL del servidor.

Para crear el servidor hemos de apoyarnos en el API de Java™ JNDI™, interfaz estándar de Java™ para múltiples servicios de directorio y de nombrado. Nosotros le pasaremos los parámetros para indicarle que vamos a trabajar con un servidor de nombres de CORBA™:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
                        "com.sun.jndi.cosnaming.CNCtxFactory");
env.put(Context.PROVIDER_URL, "iiop://localhost:3000");
Context initialContext = new InitialContext(env);
```

Una vez hecho esto creamos un objeto de la clase servidor y lo vinculamos a un nombre de la misma manera que si estuviéramos trabajando con el registro RMI:

```
HelloWorldImpl helloRef = new HelloWorldImpl();
initialContext.rebind("HelloService", helloRef );
```

3.5.2.2. Cliente RMI/IIOP.

Para el cliente Java™ partimos del interfaz de *callback* del servidor y procedemos igual que si se tratase de un servidor normal.

Implementamos una clase que derive de *javax.rmi.PortableRemoteObject*, compilamos todo y con rmic generamos los esqueletos y los cabos:

```
rmic -iiop HelloWorldCallbackImpl
```

El IDL no será necesario de generar en este caso porque nos valdrá ya con el generado en el servidor.

Usaremos la interfaz JNDI™ y la prepararemos para trabajar con un servidor de nombres de igual manera que en el servidor.

Finalmente usamos el método *lookup* para obtener una referencia remota del objeto *HelloWorld* del servidor. El método *lookup* devolverá un *object* de la que obtendremos una referencia a *Helloworld* mediante el método estático *narrow* de *PortableRemoteObject*:

```
Object objref=initialContext.lookup("HelloService");
  HelloWorld hw=(HelloWorld)PortableRemoteObject.narrow(
          objref, HelloWorld.class);
```

En este momento ya podemos usar referencia como si de un objeto local se tratara, excepto en el caso de los *casting* como hemos visto en el paso anterior:

```
hw.sayHello("Hello World!");
System.out.println(hw.getVirtualMachine());
hw.execCallback(new HelloWorldCallbackImpl());
```

3.5.2.3. Servidor IIOP.Net.

Partimos del IDL generado con el servidor RMI/IIOP y con la ayuda del compilador IDLToCLSCompiler generamos una dll que contenga los meta datos que necesitamos para comunicarnos con el cliente RMI/IIOP:

```
IDLToCLSCompiler HelloWorld HelloWorld.idl HelloWorldCallback.idl
```

Es posible que prefiramos disponer del código fuente en C# u otro lenguaje soportado por CLI en vez de un archivo .dll. Como IIOP.Net no proporciona de ninguna herramienta para generar código C# para obtener el fuente deberemos de realizar un paso más.

Existen herramientas conocidas como *decompiladores* que a partir de un ensamblado CLI, dll o ejecutable, pueden obtener código fuente en C#.

Los pasos para obtener el código fuente en C# a partir de IDL son generar una dll con IDLToCLSCompiler y posteriormente extraer de la misma el código C# con un *decompilador*.

En Internet el *decompilador* Salamander.Net genera a través de su página web código C# a partir ensamblados que se le pasan a través de un navegador.

IIOP.Net es un canal de Remoting y como en la implementación de cualquier servidor Remoting vamos proceder.

Como todos los canales .Net Remoting, IIOP.Net debe de crearse y registrarse:

```
IiopChannel channel = new IiopChannel(0);
ChannelServices.RegisterChannel(channel);
```

Escribimos una clase que implemente el servidor que derive de *MarshalByRefObject* en implemente el interfaz *HelloWorld*. Sobrescribiremos el método de *MarshalByRefObject InitializeLifetimeService* para que devuelva *null*, con lo que el tiempo de arrendamiento de los objetos de esta clase será infinito. Cuando queramos limpiar el objeto usaremos *RemotingServices.Disconnect*:

```
[SupportedInterface(typeof (HelloWorld))]
public class HelloWorldImpl : MarshalByRefObject, HelloWorld
{
      public override object InitializeLifetimeService()
      {
            return null;
      }
      ...
}
```

Para trabajar correctamente con RMI/IIOP añadiremos el atributo *SupportedInterface* con el tipo de *HelloWorld*, si no hiciésemos esto IIOP.Net le dirá a RMI/IIOP que el objeto que se le pasa es del tipo *HelloWorldImpl*.

A continuación nos comunicamos con el servidor de nombres de CORBA™:

```
CORBAInit init = CORBAInit.GetInit();
NamingContext nameService =
                  init.GetNameService("localhost", 3000);
```

Creamos un objeto de la clase servidor y lo registramos en el servicio de nombres CORBA™:

```
NameComponent[] name = new NameComponent[] {
                  new NameComponent("HelloService", "")};
nameService.rebind(name, new HelloWorldImpl());
```

3.5.2.4. Cliente IIOP.Net.

Para crear el cliente .Net necesitamos la .dll o las clases generadas en el servidor IIOP.Net a partir del IDL del servidor RMI/IIOP.

Como todos los canales .Net Remoting, IIOP.Net debe de crearse y registrarse:

```
IiopChannel channel = new IiopChannel(0);
ChannelServices.RegisterChannel(channel);
```

Los objetos que sirven para hacer *callbacks* son también objetos distribuidos, por lo tanto necesitamos proceder con ellos igual que con cualquier servidor Remoting. Al igual que con el servidor haremos una clase que derive de *MarshalByRefObject* en implemente el interfaz *HelloWorldCallback*.

Además también en esta clase vamos a sobreescribir el método de *MarshalByRefObject InitializeLifetimeService* para que devuelva *null* con lo que el tiempo de arrendamiento del objeto será infinito. Cuando queramos limpiar el objeto usaremos *RemotingServices.Disconnect*:

```
[SupportedInterface(typeof (HelloWorld))]
public class HelloWorldCallbackImpl : MarshalByRefObject,
                                      HelloWorldCallback
{
      public override object InitializeLifetimeService()
      {
            return null;
      }
      ...
}
```

También hemos de usar el atributo *SupportedInterface* con el tipo de *HelloWorldCallback*.

A continuación nos comunicamos con el servidor de nombres de CORBA™:

```
CORBAInit init = CORBAInit.GetInit();
NamingContext nameService =
                          init.GetNameService("localhost",3000);
```

Creamos un *proxy* que nos permita trabajar con el objeto *HelloWorld* del servidor:

```
HelloWorld helloWorld = (HelloWorld) nameService.resolve(name);
```

Finalmente simplemente usamos el *proxy* para trabajar como si de un objeto local se tratase:

```
Console.WriteLine(helloWorld.virtualMachine);
helloWorld.sayHello("Hello World!");
HelloWorldCallback h=new HelloWorldCallbackImpl();
helloWorld.execCallback(new HelloWorldCallbackImpl());
```

3.6. Interoperatividad con servicios web.

Un Web Service es una función (o conjunto de funciones, objeto u objetos) publicada por una aplicación informática, que puede ser accedida a través de la red por otras aplicaciones, normalmente a través de HTTP, en un esquema cliente-servidor y que provee una funcionalidad específica.

3.6.1. Introducción a los servicios web.

La idea de los servicios web es aprovechar la infraestructura ya existente para las páginas web y usarla para la comunicación entre aplicaciones. Aunque hay tipos de servicios web que pueden usarse sobre otros protocolos de Internet, como los protocolos de correo electrónico, lo normal es implementarlos sobre HTTP.

Cuando usemos HTTP como transporte de las comunicaciones de nuestros servicios web podemos beneficiarnos de toda la infraestructura ya existente para el acceso a sitios web. De esta manera podremos usar los *proxys* de HTTP que tengamos disponibles para el acceso a servidores de otras redes sin tener que realizar configuraciones adicionales. Esta facilidad hace de los servicios web una herramienta muy útil para comunicaciones entre diferentes redes.

La Arquitectura Orientada a Servicios (SOA, por sus siglas en inglés) es un diseño, en el que se basan los servicios web, del tipo Cliente/Servidor en donde una aplicación se conforma de servicios de *software* y consumidores de esos servicios.

Un servicio según la arquitectura SOA es una función sin estado, auto-contenida, que acepta una varías llamadas y devuelve una o varías respuestas mediante una interfaz bien definida. Los servicios no dependen del estado de otras funciones o procesos.

3.6.2. SOAP.

SOAP (siglas de Simple Object Access Protocol) es un protocolo estándar creado por Microsoft®, IBM® y otros, está actualmente bajo el auspicio de la W3C que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. SOAP es uno de los protocolos utilizados en los servicios Web.

Junto a SOAP tenemos el lenguaje WSDL. WSDL describe la interfaz pública a los servicios Web. Está basado en XML y describe la forma de comunicación, es decir, los requisitos del protocolo y los formatos de los mensajes necesarios para interactuar con los servicios listados en su catálogo. Las operaciones y mensajes que soporta con WSDL se describen en abstracto y se ligan después al protocolo concreto de red y al formato del mensaje.

Net Remoting es un mecanismo de comunicación implementado en el .Net Framework y en Mono. Un canal de Remoting es un objeto que transporta mensajes entre dos aplicaciones. Remoting incorpora de serie

tres canales Remoting que son TcpChannel, sobre TCP/IP; HttpChannel sobre HTTP e IpcChannel, incluido a partir de la versión 2.0 de .Net Framework y que usa *pipes* con nombre de Windows®.

Para el ejemplo de interoperatividad con SOAP en el servidor .Net vamos a usar .Net Remoting con HttpChannel y el *formateador* SOAP con lo que no necesitaremos ningún componente que no esté en Mono o en el .Net Framework.

En Mono y en la versión estándar del .Net Framework podemos consumir los servicios SOAP con .Net Remoting. El Compact Framework es la versión del .Net Framework para dispositivos limitados, como PDAs y SmartPhones, que no incluye .Net Remoting. Con el .Net Compact Framework podremos realizar clientes SOAP usando las herramientas también disponibles en .Net y Mono para consumir servicios web.

Entre los principales componentes de .Net Remoting están los sumideros de mensajes. Son objetos que procesan los mensajes, elementos de .Net Remoting que contienen la información necesaria para una llamada a procedimiento remoto. Tenemos una pila de sumideros en el cliente y otra en el servidor. Entre esos sumideros de la pila servidor de SOAP en .Net Remoting está el *SDLChannelSink*. Este sumidero a diferencia del resto no realiza ninguna transformación de información para la llamada de un procedimiento remoto. Su función es ser un sumidero transparente siempre que no se encuentre con una petición HTTP GET finalizada en *?WSDL* o *?SDL*. En este caso generará el documento WSDL que define el servicio web. Este documento es el que vamos a usar para generar los clientes desde .Net y Java™ y el servidor Java™ con Axis.

Aunque la implementación de referencia del API JAX-RPC, API para estándar de Java™ para SOAP, es la del JWSDP (Java™ Web Services Developer Pack), para realizar el servidor y el cliente con Java™ vamos a optar por Axis, la implementación de Apache. Apache Axis es la implementación más usada y también la implementación sobre la que vamos a encontrar más documentación. La versión de Apache Axis que hemos usado para el ejemplo es la 1.3.

3.6.2.1. Servidor .Net Remoting.

En .Net Remoting para implementar un servidor siempre tenemos que hacer es una clase que derive de *MarshalByRefObject* y añadir como referencia a nuestro ejecutable *System.Runtime.Remoting*:

```
public class HelloWorld : MarshalByRefObject
{
      public void SayHello(String HelloMessage)
      {
            Console.WriteLine(HelloMessage);
      }

      public string getVirtualMachine()
      {
            return Type.GetType("Mono.Runtime", false) == null ?
                  "Microsoft® .NET Runtime Execution Engine" :
                  "Mono JIT compiler";
      }
```

```
}
```

Como se puede ver la clase incorpora dos métodos. Un método llamado *sayHello* que imprime en la pantalla del servidor un mensaje que se le pasa por parámetro y un método *getVirtualMachine* que devuelve un *string* indicando la máquina virtual de .Net sobre la que corre el servidor.

Suponiendo que ya tuviésemos terminado el resto de lo necesario para obtener el documento WSDL nos encontramos que el atributo *targetNamespace* contiene por defecto un valor muy largo:

```
<definitions name="HelloWorld"
targetNamespace="http://schemas.Microsoft®.com/clr/nsassem/
                        SOAPServer/SOAPServer%2C%20Version
%3D0.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
```

Esto no debiera de preocuparnos a no ser que con él deseemos crear clientes Java™. En Java™ este valor se *mapea* como un paquete, el valor por defecto no es válido para un paquete Java™ porque contiene caracteres no permitidos. Por este motivo vamos a añadir un atributo que indique a .Net Remoting que valor debe tener este parámetro en el documento WSDL:

```
[SOAPType(XmlNamespace="HelloWorld")]
public class HelloWorld : MarshalByRefObject
...
```

Con este atributo el comienzo del documento WSDL será el siguiente:

```
<definitions name="HelloWorld" targetNamespace="HelloWorld">
```

En .Net Remoting tenemos el concepto de *Leasing*. Es el tiempo que un objeto puede permanecer publicado por .Net Remoting sin recibir ninguna petición. Si queremos que este tiempo sea infinito debemos que sobrescribir el método *InitializeLifetimeService* y hacer que devuelva *null*:

```
public override object InitializeLifetimeService()
{
      return null;
}
```

Al añadir este método a la clase *HelloWorld* y generar el WSDL nos encontramos que .Net Remoting también publica ese método. Vamos a tratar de ocultarlo creando una nueva clase que herede de *HelloWorld* dónde sobrescribir el método:

```
public class HelloWorldService : HelloWorld
{
      public override object InitializeLifetimeService()
      {
            return null;
      }
}
```

El objetivo de esta clase es crear un objeto de la clase *HelloWorldService* y publicarlo como *HelloWorld*, superclase de la anterior dónde no aparece reescrito *InitializeLifetimeService*.

Para trabajar con SOAP y .Net Remoting hay que crear un canal *HttpChannel* y registrarlo de la siguiente forma:

```
HttpChannel httpChannel = new HttpChannel(3000);
ChannelServices.RegisterChannel(httpChannel);
```

A continuación crearemos un objeto *HelloWorldService* e indicaremos a .Net Remoting que lo publique como un objeto *HelloWorld* para que no aparezca *InitializeLifetimeService* en el WSDL generado por el sumidero *SDLChannelSink* del canal *HttpChannel*:

```
HelloWorldService helloWorld = new HelloWorldService();
RemotingServices.Marshal(helloWorld, "HelloWorld", typeof
                                                    (HelloWorld));
```

En el caso que queramos que este objeto no reciba más peticiones y que el *HttpChannel* tampoco desconectaríamos el objeto y desregistraríamos el canal de .Net Remoting:

```
RemotingServices.Disconnect(helloWorld);
ChannelServices.UnregisterChannel(httpChannel);
```

Para obtener el WSDL que usaremos en el resto de desarrollo que vamos a hacer con SOAP en un navegador tecleamos:

```
http://localhost:3000/HelloWorld?WSDL
```

3.6.2.2. Cliente .Net Remoting.

Existen dos maneras de acceder a servicios SOAP con el .Net Framework y con Mono. Con el .Net SDK estándar y con Mono podemos usar las dos, .Net Remoting y Webservices. La manera de acceder mediante WebService, la única que permite el Compact Framework, se explica en el apartado siguiente.

Para generar una clase que nos permita acceder al servicio SOAP necesitaremos la herramienta SOAPsuds. Para generar esta clase en C# con el servicio que hemos creado antes el comando es:

```
SOAPsuds -url:http://localhost:3000/HelloWorld?WSDL -gc
```

Con esto obtendremos el archivo InteropNS.cs que añadiremos a un nuevo proyecto de ejecutable en el que haremos el cliente.

En el cliente al igual que en el servidor tenemos que añadir la referencia a *System.Runtime.Remoting*. Dentro del archivo InteropNS.cs tendremos una clase *HelloWorld* que deriva de *RemotingClientProxy*, clase contenida en el .Net Remoting.

Si creamos un objeto de la clase *HelloWorld*, contenida en InteropNS.cs, todas las llamadas a sus métodos serán transmitidas por SOAP al servidor en la localización por defecto *http://localhost:3000/HelloWorld*. *RemotingClientProxy* incorpora la propiedad *Url* para que podamos acceder al servidor si se encuentra en otra dirección:

```
HelloWorld helloWorld=new HelloWorld();
helloWorld.Url="http://localhost:3000/HelloWorld";
helloWorld.SayHello("Hello World");
Console.WriteLine(helloWorld.getVirtualMachine());
```

3.6.2.3. Cliente .Net Webservice.

System.Runtime.Remoting depende de *System.Reflection.Emit* para generar *proxys* que transformen llamadas a procedimientos en mensajes. Como *System.Reflection.Emit* no está disponible en el Compact Framework, solamente podremos acceder a estos servicios con la infraestructura propia de los servicios web.

WSDL.exe es una herramienta del .Net Framework y de Mono encargada de generar código fuente con el que acceder a servicios web con el protocolo SOAP:

```
wsdl http://localhost:3000/HelloWorld?WSDL
```

Con este comando obtendremos el archivo HelloWorldService.cs que añadiremos al proyecto en el que haremos el cliente.

HelloWorldService.cs contiene una clase llamada *HelloWorldService*, encargada de hacer de *proxy* del objeto del servidor, derivada de *SOAPHttpClientProtocol*, clase contenida en *System.Web.Services*, referencia que tendremos que añadir al ejecutable.

Cuando creamos un objeto *HelloWorldService* e invocamos uno de sus métodos se generará una llamada al método correspondiente del objeto servidor localizado en *http://localhost:3000/HelloWorld*. *SOAPHttpClientProtocol* incorpora la propiedad *Url* para que podamos cambiar la situación por defecto del servidor. Como vemos los *proxys* generadas por WSDL.exe y SOAPSUDS.exe de cara al programador tienen un comportamiento idéntico:

```
HelloWorldService helloWorldService = new HelloWorldService();
helloWorldService.Url = "http://localhost:3000/HelloWorld";
helloWorldService.SayHello("Hello World");
Console.WriteLine(helloWorldService.getVirtualMachine());
```

3.6.2.4. Servidor Apache Axis.

Apache Axis es una implementación de SOAP que podemos usar tanto para hacer clientes como servidores. Apache Axis implementa el servidor de SOAP mediante *servlets* por lo que también vamos a necesitar un contenedor de *servlets*.

El contenedor de *servlets* que es implementación de referencia, además de ser también el más usado es el Apache Tomcat. Apache Tomcat no está diseñado para se empotrado en otras aplicaciones, además requiere numerosos archivos jar. Vamos a usar la implementación Jetty®, que es un contenedor de *servlets* que si diseñado para ser empotrado y además es usado por varios servidores de aplicaciones J2EE™, entre los que están Geronimo, JBoss™ y Jonas. La versión de Jetty® que usaremos es la 5.1.6.

Para poder realizar el servidor necesitaremos referenciar los siguientes jar, que encontramos en Axis y en Jetty®, en nuestro *classpath*: activation.jar, axis.jar, commons-discovery-0.2, commonslogging-1.0.4, javax.servlet, jaxrpc, mail, org.mortbay.jetty,saaj.jar y wsdl4j-1.5.1.

Para implementar el servidor vamos a partir del WSDL generado por la implementación de servidor que hemos hecho en .Net Remoting. Nuestro objetivo es poder usar la implementación de .Net y la de Java™ indistintamente:

```
java org.apache.axis.wsdl.WSDL2Java --server-side
                          http://localhost:3000/HelloWorld?WSDL
```

Esto nos genera un directorio que contiene un paquete con las clases y los archivos de despliegue .wsdd necesarios para que podamos implementar el servidor. Añadimos este paquete a nuestro servidor. El API jax-rpc se basa en el API RMI. De las clases de proyecto *HelloWorld* solamente hay que modificar *HelloWorldBindingImpl*, que es la clase dónde se implementan los métodos expuestos a través de SOAP:

```
public class HelloWorldBindingImpl implements
                              HelloWorld.HelloWorldPortType
   {
    public void sayHello(java.lang.String helloMessage)
                        throws java.rmi.RemoteException {
      System.out.println(helloMessage);
   }

    public java.lang.String getVirtualMachine()
                        throws java.rmi.RemoteException
   {
      return System.getProperty("java.vm.name");
   }
}
```

Ahora expondremos el servicio *Helloworld* mediante Axis y Jetty®. Jetty® puede usarse como una aplicación Java™ normal con un archivo de configuración, pero como nuestro objetivo es usarlo de forma embebida vamos a hacer todo lo que podamos mediante programa.

Tanto Jetty® como Axis usan Commons-loggin para emitir mensajes de log. Fijamos *org.apache.commons.logging.Log* para indicarle que use la implementación de *logging org.apache.commons.logging.impl.SimpleLog*. Esta implementación saca los mensajes por pantalla. Si no se fija esta variable Commons-logging intentará emitir los mensajes de log a través de Log4J, posteriormente emitirá un error y continuará usando *SimpleLog*. El objetivo de fijar *org.apache.commons.logging.Log* es quitar este mensaje de error. Posteriormente también se fija valor a *org.apache.commons.logging.simplelog.defaultlog*, el valor *Error* le indica a *SimpleLog* que no emita mensajes que no sean errores. La implementación es:

```
static {
System.setProperty("org.apache.commons.logging.Log",
                   "org.apache.commons.logging.impl.SimpleLog");
  System.setProperty(
     "org.apache.commons.logging.simplelog.defaultlog","error");
}
```

Para atender las peticiones SOAP sobre http necesitamos un servidor HTTP, con Jetty® tenemos uno implementado en la clase *HttpServer*:

```
HttpServer server = new HttpServer();
```

HTTP para funcionar necesita un contexto, es decir, un directorio raíz para las páginas web y otro para buscar los *servlets*:

```
HttpContext context = new HttpContext();
context.setContextPath("/");
context.setResourceBase(System.getProperty("jetty.home", "."));
server.addContext(context);
```

Se requiere también de un objeto que pase las peticiones HTTP a Servlets, con Jetty® este objeto es del tipo *ServletHandler*:

```
ServletHandler servlets = new ServletHandler();
context.addHandler(servlets);
```

Ahora informamos al manejador de servlets sobre las direcciones que manejara el servidor Axis, para responder a las peticiones a */HelloWorld* registramos /*. Para registrar nuevos servicios vamos a registrar la dirección */servlet/AxisServlet*. La dirección */servlet/AxisServlet* se puede quitar una vez se haya registrado el servicio SOAP:

```
servlets.addServlet("AxisServlet", "/servlet/AxisServlet",
        "org.apache.axis.transport.http.AxisServlet");
servlets.addServlet("AxisServlet", "/*",
        "org.apache.axis.transport.http.AxisServlet");
```

Después de todos estos pasos simplemente creamos el servidor, le indicamos el puerto donde debe escuchar y lo arrancamos:

```
SocketListener listener = new SocketListener();
listener.setPort(3000);
server.addListener(listener);
server.start();
```

Con los pasos anteriores hemos arrancado el servidor, pero además es necesario desplegar el servicio. Para desplegar el servicio hemos de usar el archivo deploy.wsdd generado en el paquete *HelloWorld* con WSDL2Java. El archivo deploy.wsdd contiene como nombre del servicio SOAP generado *HelloWorldPort*, para que los clientes que hemos realizado con CLI puedan usarlo necesitamos que este servicio se llame *HelloWorld*. Esto lo arreglaremos cambiando la siguiente línea de deploy.wsdd:

```
<service name="HelloWorldPort" provider="java™:RPC" style="rpc"
use="encoded">
```

Por la siguiente línea:

```
<service name="HelloWorld" provider="java™:RPC" style="rpc"
use="encoded">
```

El último paso es usar la herramienta AdminClient de Axis y ya tendremos desplegado el servicio:

```
java org.apache.axis.client.AdminClient
-lhttp://localhost:3000/hello/servlet/AxisServlet deploy.wsdd
```

3.6.2.4. Cliente Apache Axis.

Para realizar el cliente con Axis necesitaremos referenciar los siguientes jar, que encontramos en la distribución de Axis, en el *classpath*: activation.jar, axis.jar, commons-discovery-0.2, commonslogging-1.0.4, jaxrpc, mail, saaj.jar y wsdl4j-1.5.1.

Para implementar el cliente vamos a partir del WSDL generado por la implementación de servidor que hemos hecho en .Net con Remoting. Nuestro objetivo, como ya dijimos para el servidor, es poder usar la implementación de .Net y la de Java™ indistintamente:

```
java org.apache.axis.wsdl.WSDL2Java http://localhost:3000/HelloWorld?WSDL
```

Esto nos genera un directorio que contiene un paquete con las clases necesarias para que podamos implementar el cliente. Añadimos este paquete a nuestro cliente. Para acceder al servidor ya simplemente nos quedaría pedir a la clase *HelloWorldServiceLocator* que nos devuelva una instancia de *HelloWorldPortType*. Esta instancia se usa como si de un objeto local se tratase, solo que ahora las peticiones a métodos se pasan por red hasta el objeto remoto:

```
HelloWorldPortType helloWorldPortType = new
            HelloWorldServiceLocator()
                  .getHelloWorldPort(new URL(
            "http://localhost:3000/HelloWorld"));
helloWorldPortType.sayHello("Hello World");
System.out.print(helloWorldPortType.getVirtualMachine());
```

3.6.3. XML-RPC.

XML-RPC es un protocolo de llamadas a procedimientos remotos que usa XML para codificar sus peticiones y HTTP como mecanismo de transporte. Se trata de un protocolo sencillo que solamente define un pequeño grupo de protocolos y comandos y cuya especificación no ocupa más de dos hojas.

Fue creado como parte del producto Frontier de UserLand en colaboración con Microsoft® en 1995. Microsoft® consideró que se trataba de un proyecto muy simple y comenzó a añadirle funcionalidad hasta que el estándar no fue tan sencillo y se convirtió en lo que hoy es SOAP.

Apache XML-RPC es la implementación Java™ que hemos seleccionado para el ejemplo. Como veremos más adelante se trata de una

solución muy simple que para realizar el servidor incluye un sencillo servidor de HTTP empotrado. La versión de Apache XML-RPC sobre la que está hecho el ejemplo es la 2.0.

XML-RPC.Net es una librería CLI escrita por Charles Cook que implementa el protocolo XML-RPC apoyándose en IIS y en el canal Remoting HttpChannel para implementar los servidores. Además incorpora una versión de la librería para el .Net Compact Framework posibilitando consumir servicios web con XML-RPC desde dispositivos limitados como PDAs o smartphones. La versión que usaremos de esta librería será la 0.9.2.

3.6.3.1. Servidor XML-RPC.Net.

Para usar XML-RPC.net como servidor deberemos referenciar la librería CookComputing.xmlrpc.dll que contiene la implementación de XML-RPC y System.Runtime.Remoting.dll con la implementación de .Net Remoting.

Entre los principales componentes de .Net Remoting están los sumideros de mensajes. Son objetos que procesan los mensajes, elementos de .Net Remoting que contienen la información necesaria para una llamada a procedimiento remoto. Tenemos una pila de sumideros en el cliente y en el servidor. El sumidero encargado de *serializar* y *deserializar* los mensajes es el *Formatter*. El servidor XML-RPC.Net implementa los sumideros *Formatter* y aprovecha para el resto de la pila servidor de .Net Remoting la infraestructura de SOAP.

Entre el cliente y el servidor de xml-rpc.net deberemos de compartir un interfaz:

```
[XmlRpcUrl("http://localhost:3000/HelloWorld")]
public interface IHelloWorld
{
      [XmlRpcMethod("HelloWorld.SayHello")]
      string SayHello(string HelloMessage);

      [XmlRpcMethod("HelloWorld.getVirtualMachine")]
      string getVirtualMachine();
}
```

El atributo *XmlRPCUrl* para indicar el lugar dónde por defecto estará situado el servidor. En el caso que nuestros métodos deban de incluir en el nombre puntos u otros símbolos que C# no permita deberemos usar el atributo *XmlRPCMethod* para indicárselo a XML-RPC.Net. Todos los métodos en XML-RPC deben retornar algo.

Como XML-RPC.Net aprovecha la infraestructura de .Net Remoting el servidor lo implementaremos mediante una clase que derive de *MarshalByRefObject*. Además con XML-RPC.Net también tendremos que implementar el interfaz que definimos antes:

```
public class HelloWorld : MarshalByRefObject, IHelloWorld
{
      public override object InitializeLifetimeService()
      {
            return null;
      }
      [XmlRpcMethod("HelloWorld.SayHello")]
      public string SayHello(string HelloMessage)
      {
            Console.WriteLine(HelloMessage);
            return "";
      }
      [XmlRpcMethod("HelloWorld.getVirtualMachine")]
      public string getVirtualMachine()
      {
            return Type.GetType("Mono.Runtime", false) == null ?
                  "Microsoft® .NET Runtime Execution Engine" :
            "Mono JIT compiler";
      }
}
```

En .Net Remoting tenemos el concepto de *Leasing*. *Leasing* es el tiempo que un objeto puede permanecer publicado por .Net Remoting sin recibir ninguna petición. Si queremos que este tiempo sea infinito debemos que sobrescribir el método *InitializeLifetimeService* y hacer que retorne *null* como se puede ver en la clase *HelloWorld*.

Para publicar este objeto es crear un canal *HttpChannel* indicando que en lugar de usar los sumideros estándar SOAP para formatear los mensajes se usarán los del XML-RPC.Net. Posteriormente registraremos el canal en .Net Remoting:

```
ListDictionary channelProperties = new ListDictionary();
channelProperties.Add("port", 3000);
HttpChannel httpChannel = new HttpChannel(channelProperties,
                        new XmlRpcClientFormatterSinkProvider(),
                  new XmlRpcServerFormatterSinkProvider(null, null));
ChannelServices.RegisterChannel(httpChannel);
```

El único paso que nos resta es crear un objeto *HelloWorld* e indicarle a .Net Remoting que lo publique como un objeto *IHelloWorld*:

```
HelloWorld helloWorld = new HelloWorld();
RemotingServices.Marshal(helloWorld, "HelloWorld",
                                  typeof (IHelloWorld));
```

En el caso que queramos que este objeto y el *HttpChannel* no reciban más peticiones desconectaríamos el objeto y desregistraríamos el canal de Remoting:

```
RemotingServices.Disconnect(helloWorld);
ChannelServices.UnregisterChannel(httpChannel);
```

3.6.3.2. Cliente XML-RPC.Net.

XML-RPC.Net no usa .Net Remoting en la parte del cliente, por este motivo solamente necesitamos referenciar la librería CookComputing.xmlrpc.dll que contiene la implementación de XML-RPC.

Para comunicar con el servidor hemos de crear un objeto *proxy* que pase las peticiones del cliente al servidor. XML-RPC.Net nos da la opción de crearlo nosotros o dejar que lo cree él en tiempo de ejecución. Para crearlo él en tiempo de ejecución usa la librería *system.reflection.emit* que no está disponible en el Compact Framework. Vamos a escoger en el ejemplo de este apartado que lo haga él y en la versión para Compact Framework se explicará la otra versión.

En el cliente hemos de añadir el interfaz que hemos definido en el servidor. En esta versión solamente debemos de indicar a XML-RPC.Net que cree el objeto *proxy* que necesitamos para comunicarnos con el servidor a partir del interfaz que tienen en común cliente y servidor, este objeto implementará *IHelloWorld* y derivará de *XmlRpcClientProtocol*. Si queremos indicarle una dirección de servidor que no sea la que lleva el interfaz por defecto le daremos valor a la propiedad *Url* de *XmlRpcClientProtocol*:

```
IHelloWorld ihelloWorld = (IHelloWorld)
                  XmlRpcProxyGen.Create(typeof (IHelloWorld));
((XmlRpcClientProtocol) ihelloWorld).Url =
                  "http://localhost:3000/HelloWorld";
```

Este objeto lo usaremos como si de cualquier un objeto local se tratase:

```
ihelloWorld.SayHello("Hello World");
Console.WriteLine(ihelloWorld.getVirtualMachine());
```

3.6.3.3. Cliente XML-RPC.Net CF.

XML-RPC.Net proporciona una librería reducida para el Compact Framework igual a la versión estándar eliminando todas las referencias a *system.reflection.emit* que debemos de referenciar en nuestra aplicación. Al eliminar *system.reflection.emit* esta librería carece de la clase *XmlRPCProxyGen* con la que se pueden crear los *proxys* a partir del interfaz.

En el Compact Framework hemos de crear el *proxy* nosotros con una clase que derive de *XmlRpcClientProtocol* e implemente *IHelloWorld*. La implementación de cada método no es más que llamar al método *Invoke* de *XmlRpcClientProtocol* indicándole el nombre del método remoto a llamar y sus parámetros en un *array*:

```
public class HelloWorld : XmlRpcClientProtocol, IHelloWorld
{
      public string SayHello(string HelloMessage)
      {
            return (string) base.Invoke("SayHello",
                                    new object[] {HelloMessage});
      }
      public string getVirtualMachine()
      {
            return (string) base.Invoke("getVirtualMachine",
                                                new object[] {});
      }
}
```

Para acceder a los procedimientos del servidor es suficiente con crear un objeto de esta clase. Si queremos indicarle una dirección de servidor que

no sea la que lleva el interfaz por defecto le daremos valor a la propiedad *Url* de *XmlRpcClientProtocol*:

```
HelloWorld helloWorld = new HelloWorld();
helloWorld.Url = "http://localhost:3000/HelloWorld";
```

Este objeto lo usaremos como si de cualquier un objeto local se tratase y el se encargará de llamar a los procedimientos del servidor:

```
helloWorld.SayHello("Hello World");
Console.WriteLine(helloWorld.getVirtualMachine());
```

3.6.3.4. Servidor Apache XML-RPC.

Apache XML-RPC implementa el *servlet* *XmlRpcServer* con el que podemos desplegar servicios web XML-RPC en un contenedor de *servlets* como Jetty® o Apache Tomcat. También encontramos dentro de la distribución el servidor empotrado HTTP *WebServer* con el que podremos publicar nuestros servicios sin necesidad de un contenedor de *servlets*.

Aunque para el ejemplo hemos usado el servidor HTTP empotrado por simplicidad conviene aclarar que esta solución presenta la limitación consistente en sólo poder albergar servicios web XML-RPC que puedan estar en el mismo *path* del servidor. El servidor empotrado no interpreta el *path* en las direcciones URL de manera que para él es lo mismo la dirección *http://localhost:3000/HelloWorld* que la dirección *http://localhost:3000/RPC2*. Los servicios web los distingue mediante un prefijo que añade a cada método del mismo seguido de un punto que indicamos al registrar el servicio en el servidor. Si damos el valor *$default* a un servicio web al registrarlo los métodos de este servicio no tendrán prefijo.

Con el servidor empotrado solamente tenemos que añadir al *classpath* los jar xmlrpc-2.0.jar y commons-codec-1.3.jar.

El servicio web se implementa mediante una clase, todos los métodos que queramos que puedan ser accedidos mediante XML-RPC deben de contener únicamente parámetros de tipos especificados en el protocolo y además nunca devolver *void*:

```
public class HelloWorld {
      public String getVirtualMachine() {
            return System.getProperty("Java.vm.name");
      }

      public String SayHello(String HelloMessage) {
            System.out.println(HelloMessage);
            return "";
      }
}
```

Para hacer un servicio creamos un objeto *WebServer* indicando el puerto dónde recibirá las peticiones el servidor, añadimos al mismo como servicio web una instancia de la clase que implementa el servicio y un nombre que hará de prefijo de todos los procedimientos del servidor, como vimos antes, y arrancamos:

```
WebServer server = new WebServer (3000);
server.addHandler ("HelloWorld", new HelloWorld());
server.start ();
```

3.6.3.5. Cliente Apache XML-RPC.

Con el servidor empotrado solamente tenemos que añadir al *classpath* los jar xmlrpc-2.0.jar y commons-codec-1.3.jar.

En la parte cliente disponemos de un único objeto *XmlRpcClient* al que se le pasa en el constructor la dirección del servidor:

```
XmlRpcClient client=new
            XmlRpcClient("http://localhost:3000/helloworld");
```

Luego, para terminar, mediante el método *execute* llamamos a los métodos del servidor. Este procedimiento recibe dos parámetros, el nombre del método y un vector con los parámetros:

```
Vector params=new Vector();
params.add(new String("Hello World"));
client.execute("HelloWorld.SayHello",params);

params.clear();
System.out.println(
client.execute("HelloWorld.getVirtualMachine",params));
```

3.6.4. Hessian.

Hessian es un servicio web binario, es decir, no usa XML para transportar la información. Hace que podamos tener servicios web usables sin necesidad de grandes *frameworks* y sin tener que aprender una sopa de letras de protocolos. Al ser un protocolo binario es apropiado para enviar datos binarios sin tener que extender el protocolo con añadidos.

Hessian fue desarrollado por Caucho Tenchnology, Inc. para su servidor de aplicaciones Resin™. El servidor de aplicaciones Spring Framework también soporta este protocolo. Caucho Tenchnology, Inc. ofrece implementaciones de Hessian en Java™ y Python bajo licencia Apache. Tenemos también disponibles bajo licencias de fuente abierta implementaciones en C++, C#, PHP y Ruby. Vamos usar la implementación HessianC# para CLI, que recientemente ha cambiado su licencia de GPL a LGPL.

3.6.4.1. Servidor HessianC#.

Para el servidor vamos a usar HessianC# 1.2.1, esta versión es la primera que incorpora un servidor web empotrado y también la primera bajo licencia LGPL.

Para usar Hessian deberemos referenciar la libreria Hessiancsharp.dll que contiene la implementación de Hessian. Entre el cliente y el servidor de Hessian deberemos de compartir un interfaz:

```
public interface IHelloWorld
{
      void SayHello(string HelloMessage);
      string getVirtualMachine();
}
```

Para implementar el servicio heredamos de este interfaz:

```
public class HelloWorld :IHelloWorld
{
      public void SayHello(string HelloMessage)
      {
            Console.WriteLine(HelloMessage);
      }
      public string getVirtualMachine()
      {
            return Type.GetType("Mono.Runtime", false) == null ?
                  "Microsoft® .NET Runtime Execution Engine" :
                   "Mono JIT compiler";
      }
}
```

Como es un servicio web necesitamos un servidor HTTP, usaremos el empotrado de HessianC#. Al crear el servidor le indicamos el puerto donde debe escuchar, la dirección del servidor, el tipo de servidor que es y el objeto que responde a las peticiones que se la hagan:

```
CWebServer web = new CWebServer(3000, "/helloworld",
                        typeof (IHelloWorld),new HelloWorld());
```

Si introducimos mayúsculas en la ruta del servidor este no funcionará. Para que el servidor acepte peticiones de cualquier cliente sin pasarlas por ningún tipo de filtro hay que pasarlo al modo no paranoico. Por defecto los servidores se crean en modo paranoico. Posteriormente arrancamos el servidor:

```
web.Paranoid=false;
web.Run();
```

3.6.4.2. Cliente HessianC#.

HessianC# nos da dos opciones para comunicar con el cliente, una mediante *proxys* y otra mediante un objeto que realiza llamadas remotas indicándole la meta información de cada procedimiento y un *array* con los parámetros del mismo. Para crear los *proxys* HessianC# utiliza *System.Reflection.Emit*, esta librería no está en el Compact Framework. Vamos a explicar como realizar llamadas a través de *proxys* en este apartado y dejar el otro método para el cliente del Compact Framework.

Para usar Hessian deberemos referenciar la librería Hessiancsharp.dll que contiene la implementación de Hessian y copiar el interfaz que implementa el servicio publicado en el servidor.

El *proxy* lo creamos con una factoría de *proxys*. Las factorías de *proxys* incorporan un método *create* al que le tenemos que pasar la dirección del servicio y el tipo del interfaz del servicio web:

```
CHessianProxyFactory factory = new CHessianProxyFactory();
IHelloWorld ihelloWorld = (IHelloWorld)
factory.Create(typeof (IHelloWorld),
                              "http://127.0.0.1:3000/helloworld");
```

Simplemente queda por usar el objeto *proxy* a través de los procedimientos del interfaz para realizar las llamadas a los procedimientos remotos:

```
ihelloWorld.SayHello("Hello World");
Console.WriteLine(ihelloWorld.getVirtualMachine());
```

3.6.4.3. Cliente HessianC# Compact Framework.

Para usar Hessian con el Compact Framework deberemos referenciar la librería Hessianmobileclient.dll que contiene la implementación de Hessian y copiar el interfaz que implementa es servicio publicado en el servidor. Para llamar a los métodos del servidor tenemos que crear un objeto *CHessianMethodCaller*, este objeto tiene de parámetros una factoría de *proxys* y la dirección del servidor:

```
CHessianProxyFactory factory = new CHessianProxyFactory();
CHessianMethodCaller cHessianMethodCaller = new
CHessianMethodCaller(factory,
                  New Uri("http://127.0.0.1:3000/helloworld"));
```

Para llamar a los métodos del servicio usaremos la meta información de cada método del interfaz del servicio en combinación de un *array* con los valores de los parámetros:

```
cHessianMethodCaller.DoHessianMethodCall(new object[]
                  {"Hello World"},
                  typeof (IHelloWorld).GetMethod("SayHello"));

Console.WriteLine(cHessianMethodCaller.DoHessianMethodCall(
            new object[] {},
            typeof(IHelloWorld).GetMethod("getVirtualMachine")));
```

3.6.4.4. Servidor Hessian.

El servidor de Hessian está implementado como un *servlet*, por tanto también vamos a necesitar un contenedor de *servlets*. El contenedor de *servlets* que es implementación de referencia, además de ser también el más usado es Apache Tomcat. Apache Tomcat no está pensado para se empotrado en otras aplicaciones, además depende de un gran número de jars. Vamos a usar la implementación Jetty®, que es un contenedor de *servlets* diseñado para ser empotrado en otras aplicaciones y usado por varios servidores de aplicaciones J2EE™, entre los que están Geronimo, JBoss™ y Jonas.

La versión de Jetty® que usaremos es la 5.1.6. Para poder realizar el servidor necesitaremos referenciar los siguientes jar, que encontramos en Hessian y en Jetty®, en nuestro *classpath*: commonslogging-1.0.4, javax.servlet, org.mortbay.jetty, saaj.jar y hessian-3.0.13.jar.

Nuestro objetivo es poder usar la implementación de .Net y la de Java™ indistintamente, para ello partiremos del interfaz que hemos

realizado en C# y lo traduciremos a Java™. Los tipos del retorno y de los parámetros los traduciremos según la equivalencia entre Java™ y C#:

```
public interface IHelloWorld {
      public String getVirtualMachine();
      public void SayHello(String HelloMessage);
}
```

Hacemos una clase que implemente este interfaz para realizar el servicio web:

```
public class HelloWorld implements IHelloWorld {
      public String getVirtualMachine() {
            return System.getProperty("Java.vm.name");
      }
      public void SayHello(String HelloMessage) {
            System.out.println(HelloMessage);
      }
}
```

Jetty® usa Commons-logging para emitir mensajes de log. Fijamos *org.apache.commons.logging.Log* para indicarle que use la implementación de *logging org.apache.commons.logging.impl.SimpleLog*. Esta implementación saca los mensajes por pantalla. Si no se fija esta variable Commons-logging intentará emitir los mensajes de *log* a través de Log4J, posteriormente emitirá un error y continuará usando *SimpleLog*. El objetivo de fijar *org.apache.commons.logging.Log* es quitar este mensaje de error. Posteriormente también le asignaremos el valor *Error* a *org.apache.commons.logging.simplelog.defaultlog*, así indicamos a *SimpleLog* que no emita mensajes que no sean errores. La implementación es:

```
static {
System.setProperty("org.apache.commons.logging.Log",
                   "org.apache.commons.logging.impl.SimpleLog");
System.setProperty(
      "org.apache.commons.logging.simplelog.defaultlog","error");
}
```

Para atender las peticiones Hessian con HTTP necesitamos es un servidor HTTP, con Jetty® tenemos uno implementado en la clase *HttpServe*r:

```
HttpServer server = new HttpServer();
```

HttpServer para funcionar necesita un contexto, es decir, un directorio raíz para las páginas web y otro para buscar los *servlets*:

```
HttpContext context = new HttpContext();
context.setContextPath("/");
context.setResourceBase(System.getProperty("jetty.home", "."));
server.addContext(context);
```

Posteriormente necesitamos un objeto que pase las peticiones HTTP a *servlets*, con Jetty® este objeto debe ser del tipo *ServletHandler*:

```
ServletHandler servlets = new ServletHandler();
context.addHandler(servlets);
```

Ahora informamos al manejador de *servlets* sobre las direcciones que manejara el servidor Hessian, para responder a las peticiones registramos */helloworld*. Para indicarle el tipo del servidor al *servlet* fijamos el parámetro *home-class* e indicamos el interfaz con el parámetro *home-api*:

```
ServletHolder servletholder=servlets.addServlet("HelloWorld",
                                                "/helloworld",
                  "com.caucho.hessian.server.HessianServlet");
servletholder.setInitParameter("home-class","HelloWorld");
servletholder.setInitParameter("home-api","IHelloWorld");
```

Una vez hechos todos estos pasos indicaremos al servidor el puerto donde debe de escuchar y lo arrancamos:

```
SocketListener listener = new SocketListener();
listener.setPort(3000);
server.addListener(listener);
server.start();
```

3.6.4.5. Cliente Hessian.

En el cliente Java™ simplemente tenemos que añadir en el *classpath* hessian-3.0.13.jar. Los clientes Hessian y HessianC# son pacticamente idénticos. En Hessian tenemos una factoría de *proxys*. Las factorías de *proxys* incorporan un método *create* al que le tenemos que pasar la dirección del servicio y el tipo del interfaz del servicio web:

```
HessianProxyFactory factory = new HessianProxyFactory();
IHelloWorld iHelloWorld = (IHelloWorld) factory.create(
      IHelloWorld.class, "http://127.0.0.1:3000/helloworld");
```

Con el objeto proxy a través de los procedimientos del interfaz realizamos las llamadas a los procedimientos remotos:

```
iHelloWorld.SayHello("Hello World");
System.out.println(iHelloWorld.getVirtualMachine());
```

3.7. Interoperatividad con Middleware orientado a mensajes.

El Middleware Orientado a Mensajes, también conocido como Middleware de Mensajes o MOM, es un software que provee de un interfaz entra aplicaciones, permitiéndolas enviar y recibir datos, mensajes, entre ellas de manera asíncrona. Los mensajes pueden ser guardados en una cola hasta que el receptor esté en condiciones de procesarlos. El MOM libera a las aplicaciones de tener que ocuparse de que los mensajes son recibidos.

3.7.1. Introducción al Middleware orientado a mensajes.

El MOM y el correo electrónico tienen una funcionalidad de transporte similar. La diferencia está en que el MOM comunica aplicaciones y el software de correo electrónico está pensado para comunicar personas.

Las principales ventajas de las comunicaciones basadas en mensajes son las capacidades de guardar, *enrutar* o adaptar el mensaje mientras es enviado.

La mayoría de sistemas MOM proporcionan un sistema de almacenado de datos. Esto libera al cliente y al receptor de estar conectados a la vez. Esto permite que si el receptor falla por cualquier motivo el servidor pueda continuar trabajando, los mensajes se acumularán hasta que el cliente esté en condiciones de recibirlos.

La capacidad de *enrutar* mensajes dentro del *middleware* MOM hace posible enviar un mensaje a varios receptores. La capacidad de los sistemas MOM para adaptar mensajes permite que se puedan enviar mensajes entre aplicaciones no necesariamente iguales, esto es lo que permite la interoperatividad mediante sistemas MOM. La capacidad de adaptar mensajes junto a la capacidad de *enrutado* permiten que enviemos un mensaje a aplicaciones heterogéneas.

Un manejador de mensajes o agente de mensajes es necesario con los sistemas MOM. Este agente trabaja con sistemas de transporte cliente/servidor e implementa la inteligencia de *enrutado* y la adaptación de mensajes. Un sistema de reglas analiza los mensajes y decide que aplicaciones los recibirán, después un sistema de formateo adaptará el mensaje a las necesidades del receptor.

Agente de mensajería.

El hecho de tener que añadir un elemento nuevo en los sistemas MOM causa una reducción en del desempeño y la consistencia, y también hace el sistema más difícil de mantener.

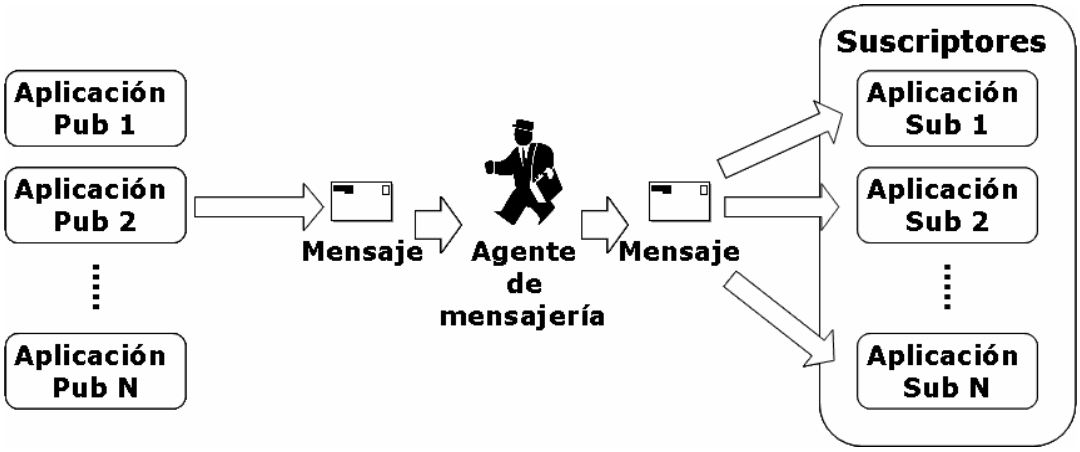
Existen muchas comunicaciones entre aplicaciones que por su naturaleza deben ser *síncronas*. Los sistemas MOM por definición son asíncronos, aunque pueden implementar mecanismos de calidad del servicio, QOS según sus siglas en inglés, para hacer comunicaciones seudo-síncronas.

Los principales mecanismos de mensajera son publicación/suscripción y comunicación punto a punto mediante colas.

3.7.1.1. Modelo publicación/suscripción.

Se suelen abreviar con *pub/sub*. En el modelo publicación/suscripción un productor envía un mensaje a un canal virtual.

Los consumidores pueden subscribirse a dicho canal, con lo que recibirían una copia del mensaje; todos los mensajes enviados a ese canal son entregados a todos los receptores. En este modelo se conoce al productor como publicador y al consumidor como subscriptor. Un aspecto importante en este modelo es que el publicador no conoce nada acerca de los subscriptores, no sabe donde se encuentran, ni cuantos hay ni lo que hacen con los mensajes. Asimismo, los receptores tienen que consumir los mensajes tal cual les llegan.



Modelo Pub/Sub.

Los aspectos importantes de este modelo son los siguientes:

- No existe acoplamiento entre productores y consumidores, pueden ser añadidos dinámicamente.
- Cada subscriptor recibe su propia copia del mensaje.
- Los subscriptores reciben el mensaje sin tener que solicitarlo. Los mensajes publicados son automáticamente entregados a los subscriptores.

Este modelo es el proporcionado por el servicio de eventos de CORBA™, XMLBlaster y JMS, que denomina a estos canales tópicos.

3.7.1.2. Modelo Punto a Punto.

El modelo punto a punto se basa en otro esquema. Los productores envían mensajes a través de canales virtuales
también llamados colas. Aquí se llaman también a los productores como emisores y a los consumidores como receptores.

Se trata de un modelo en el cual los receptores chequean la cola para ver si han recibido algún mensaje, contrariamente a lo que sucedía en el anterior modelo. En una cola puede haber más de un receptor esperando mensajes, aunque solamente uno de ellos va a consumir cada mensaje.

Modelo PtP.

El receptor en algunas implementaciones es capaz de examinar los mensajes pendientes antes de consumirlos, de forma que puede descartar alguno de ellos. Esta es una característica diferenciadora del anterior modelo, además de las que explicamos a continuación:

- Los mensajes se intercambian a través de colas.
- Cada mensaje se entrega a un solo receptor.
- Los mensajes llegan ordenados, a medida que se consumen se van eliminando de la cola.
- No existe acoplamiento entre emisores y receptores, se pueden añadir dinámicamente.

Este modelo es implementado por MSMQ, XMLBlaster y JMS.

3.7.2. MSMQ.

La cola de mensajes de Microsoft® o MSMQ es una tecnología desarrollada para el sistema Windows®. Es un *middleware* orientado a mensajes que implementa el modelo de cola. Podemos acceder a MSMQ a través de un API C, un interfaz COM y el paquete System.Messaging de .Net.

Microsoft® incorporará MSMQ en su próximo *framework* de mensajería y RPC Windows Communication Foundation, anteriormente conocido como Avalon, con el sistema Windows Vista.

El agente de mensajes con MSMQ es un servicio de Windows que hemos de instalar para poder trabajar con colas. Se llama Message Queuing.

Las colas de mensajes en Windows las podemos crear y borrar en el Administrador de Sistema de Windows o a través de código.

Los tipos de colas básicos de MSMQ son las colas privadas y las colas públicas. Las colas públicas son las que están publicadas en el directorio activo de Windows®, esto permite que sean accedidas desde otros equipos. Las colas privadas solamente son accedidas desde el propio directorio.

3.7.2.1. MSMQJava.

MSMQJava es un envoltorio al API de C hecho con JNI de MSMQ, en el ejemplo se ha usado la versión de 10 de mayo de 2005 que se puede descargar en el *blog* All About Interop de MSDN[5].

MSMQJava incluye una librería nativa JNIMSMQ.dll y tres clases Java™ que acceden a esta librería dentro del paquete *ionic.Msmq*.

Ionic.Msmq está compuesto por tres clases, una para manejar las colas (*Queue*), otra para contener los mensajes (*Message*) y una última para lanzar excepciones (*MessageQueueException*).

La clase *Queue* lanza excepciones *MessageQueueException* en todos sus métodos y en el constructor cuando una operación no puede realizarse.

Para abrir una cola usaremos el constructor. En el código se trata de abrir la cola privada *MsmqTest*:

```
queue = new Queue("DIRECT=OS:.\\private$\\MsmqTest");
```

Si lo que queremos es crear una cola usamos el método estático *create* en el que le indicamos el nombre y si la cola es *transacional* o no:

```
queue=Queue.create(".\\private$\\TSS_MsmqTest","Created by
                                              HelloMSMQ",false);
```

Para borrar tenemos el método *delete*:

```
Queue.delete(".\\private$\\MsmqTest");
```

El envío de mensajes se realiza con el método *send*, que lo tenemos sobrecargado para enviar cadenas:

```
queue.send("Hello World!");
```

Con MSMQJava hay que pedir los mensajes a la cola, se hace con los métodos *peek* y *receive*. La diferencia entre *peek* y *receive* es que *peek* no elimina un mensaje de la cola cuando se lee. Podemos limitar el tiempo de espera indicándolo en milisegundos:

```
Message rmsg=queue.receive(1000);
```

Cuando dejemos de usar la cola la tendremos que cerrar:

```
queue.close();
```

3.7.2.2. System.Messaging.

System.Messaging es el espacio de nombres a través del que podremos acceder a MSMQ desde .Net. Incorpora más funcionalidades de las que podemos obtener con MSMQJava. A continuación vamos a exponer

[5] http://blogs.msdn.com/dotnetinterop/

como hacer con *System.Messaging* todo lo que se puede hacer con MSMQJava.

Para poder usar *System.Messaging* hemos de referenciar el ensamblado System.Messaging.dll en nuestros proyectos.

System.Messaging tiene el concepto de *formateadores*. Los *formateadores serializan* los cuerpos de los mensajes. .Net dispone de tres, dos de ellos basados en los *serializadores* binario y XML que vienen con .Net. Disponemos de un tercer *formateador* para trabajar con los clientes que usen la interfaz ActiveX® (COM).

No tenemos ningún *formateador* que interprete o escriba los mensajes para trabajar con MSMQJava. El *formateador* que necesitamos simplemente deberá de leer y escribir los mensajes tal cual se pasan por MsMq.

IMessageFormatter es la interfaz que tiene que implementar todos lo *formateadores*. Contiene tres métodos y extiende el interfaz *ICloneable*.

Para tener un *formateador* con el que poder trabajar con clientes MSMQJava tendremos que hacer una clase que implemente el interfaz *IMessageFormatter*. Los métodos del interfaz *IMessageFormatter* son:

- *Read* lee los mensajes de la cola:

```
public object Read(Message message)
{
      Stream stm = message.BodyStream;
      StreamReader reader = new StreamReader(stm);
      return reader.ReadToEnd();
}
```

- *CanRead* informa al *System.Messaging* si el mensaje se puede leer o no, como este formateador no modifica los mensajes podrá leerlos siempre:

```
public bool CanRead(Message message)
{
      return true;
}
```

- *Write* escribe el mensaje en la cola para enviarlo:

```
public void Write(Message message, object obj)
{
      byte[] buff= Encoding.UTF8.GetBytes(obj.ToString());
      Stream stm = new MemoryStream(buff);
      message.BodyStream = stm;
}
```

Para que *System.Messaging* use este *formateador* tenemos que indicárselo al objeto MessageQueue, la clase con la que se tratan las colas:

```
messageQueue.Formatter = new SimpleFormater();
```

Con MessageQueue tenemos más funcionalidad que con la clase *Queue* de MSMQJava, pero como ya se ha comentado antes, se va a contemplar únicamente la funcionalidad que también tenemos disponible en Java™.

Para consultar si una cola existe tenemos el método estático *Exists*:

```
MessageQueue.Exists(@".\private$\MsmqTest")
```

Para abrir una cola usaremos el constructor. En el código se trata de abrir la cola privada *MsmqTes*t:

```
messageQueue= new MessageQueue(@".\private$\MsmqTest");
```

Si lo que queremos es crear una cola usamos el método estático *create* en el que le indicamos el nombre y si la cola es transacional o no:

```
messageQueue=MessageQueue.Create(@".\private$\MsmqTest",false);
```

Para borrar tenemos el método *delete*:

```
MessageQueue.Delete(@".\private$\MsmqTest");
```

El envío de mensajes se realiza con el método *send*, que lo tenemos sobrecargado para enviar cadenas:

```
MessageQueue.Send("Hello World!");
```

Para pedir los mensajes a la cola tenemos métodos *Peek* y *Receive*. La diferencia entre *Peek* y *Receive* es que *Peek* no elimina un mensaje de la cola cuando se lee. Podemos limitar el tiempo de espera indicándolo con *TimeSpan*:

```
Message rmsg=messageQueue.Receive(new TimeSpan(0,0,0,1,0));
```

Cuando dejemos de usar la cola la tenemos que cerrar:

```
messageQueue.Close();
```

3.7.3. Servicio de eventos de CORBA™.

CORBA™ es más que la especificación multiplataforma de la que hablamos en el apartado 3.5.1. , también define servicios habitualmente necesarios como seguridad y transacciones.

Para cada servicio del estándar CORBA™ la OMG™ provee una especificación formal descrita en OMG™ IDL (como invocar cada operación dentro de un objeto) y su semántica en lenguaje ingles (que hace cada operación y las reglas de comportamiento).

El servicio de eventos CORBA™ permite desacoplar la comunicación entre objetos. En el servicio de eventos se definen dos roles para los objetos, el rol de proveedor (*Supplier*) que corresponde a los objetos que

producen información que es tratada por los objetos que realizan el rol consumidor (*Consumer*). La información pasada entre proveedores y consumidores se agrupo en conjuntos tratados de forma atómica que denominaremos eventos. Los eventos viajan de los proveedores a los consumidores a través de comunicaciones CORBA™ estándar.

El servicio de eventos CORBA™ es un *middleware* orientado a mensajes que usa CORBA™ con infraestructura para el transporte de los mensajes a los que denomina eventos.

Existen dos maneras estándar de abordar la manera de entablar la comunicación con eventos entre proveedores y consumidores, además también hay dos maneras ortogonales de enviar la información.

Las dos manera estándar de entablar comunicación con eventos se llamar el modelo de inyección y el modelo de extracción. El modelo de inyección permite al proveedor comenzar a enviar la información a los consumidores, en el modelo de extracción son los consumidores los que reclaman a los proveedores que les pasen información. En el modelo de inyección es el proveedor el que toma la iniciativa mientras que en el modelo de extracción es el consumidor.

Para la realización de los siguientes ejemplos vamos a usar con Java™ la implementación del servicio de eventos de Jacorb 1.4.1. junto al ORB de Sun™ incluido son el JDK™. Para la realización de los ejemplos correspondientes con C# usaremos el canal de eventos realizado como parte de este proyecto y la implementación de IIOP™ IIOP.Net 1.7.1 que ya se usó en el apartado 3.5.1.

Con el fin de tener unos parámetros de línea de comando homogéneos en .Net y Java™ nos hemos ayudado de la clase *ORBHelper* que hemos obtenido del artículo *Using Multiple Vendor's ORBs with Name Services*[6].

En el apartado 4.1.1. se describe más extensamente el servicio CORBA™ EventChannel y se documentan los interfaces OMG™ IDL del mismo.

En todos los ejemplos necesitaremos tener arrancado un servicio CORBA™ de servidor de nombres. Java™ incluye una implementación del servidor de nombres que arrancaremos con el siguiente comando:

```
orbd -ORBInitialPort 3000
```

3.7.3.1. Canal de Eventos Jacorb.

La clase *EventChannelImpl* contenida en el paquete *org.jacorb.events* de Jacorb implementa un canal de eventos de acuerdo a la especificación del servicio EventChannel de CORBA™. Dicha implementación se encuentra en el archivo jacorb.jar que hemos de añadir al *classpath*.

[6] Eric Y. Theriault, 2004 (http://www.eyt.ca/CORBA/nameservice.html).

Construir un canal de eventos con la implementación de Jacorb es tan sencillo como crear un servidor CORBA™ con un objeto de la clase *EventChannelImpl* de sirviente.

Comenzamos por inicializar un *ORB*, el *ORB* es el objeto encargado en un servidor de recibir las peticiones sobre objetos de la red, identificado cada uno a través de una referencia CORBA™ y pasárselas al *POA* que alberga al objeto sirviente correspondiente que implementa la funcionalidad:

```
ORB orb = ORB.init(args, null);
```

POA es una tipo de adaptador de objeto, en CORBA™ los adaptadores de objeto son los objetos encargados de pasar una petición que les transmite un ORB a un objeto sirviente. Para trabajar con *POA*s hay que obtener siempre la referencia al *POA* raíz, que es el gestionado por el *ORB*:

```
POA rootpoa = (POA) orb.resolve_initial_references("RootPOA");
```

Un objeto *POAManager* está asociado con uno o más *POA*s. Es el objeto encargado de controlar el estado de procesamiento de los *POA*s. Cuando se crea un *POAManager* este se encuentra en estado *Holding*, estado en que todas las peticiones entrantes son encoladas, para que el *POA* procese las peticiones es necesario pasarlo al estado *Active* con el siguiente código:

```
rootpoa.the_POAManager().activate();
```

A continuación obtendremos una referencia al servidor de nombres CORBA™ con el que publicaremos el canal de eventos, esta referencia la obtiene el *ORB* a través de los parámetros de la línea de comandos:

```
org.omg.CORBA.Object objRef = orb
                  .resolve_initial_references("NameService");
```

Con el método *narrow* pasamos de una referencia CORBA™ genérica *org.omg.CORBA.Object* a una del tipo concreto que necesitamos:

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Creamos un objeto de la clase *EventChannelImpl*, que es el sirviente que implementa la funcionalidad:

```
EventChannelImpl eventChannelImpl = new
                                EventChannelImpl(orb,rootpoa);
```

Servant_to_reference asocia un determinado objeto sirviente a una referencia, obtenemos la referencia *org.omg.CORBA.Object* genérica y con *narrow* la convertimos a una de tipo *EventChannel*:

```
org.omg.CORBA.Object ref = rootpoa
                          .servant_to_reference(eventChannelImpl);
EventChannel href = EventChannelHelper.narrow(ref);
```

Con el método *rebind* de la referencia al servidor de nombres asociamos un nombre a la referencia del canal de eventos:

```
NameComponent path[] = ncRef.to_name("EventChannel");
ncRef.rebind(path, href);
```

Por ultimo llamamos al método *run*, bucle principal de un programa servidor CORBA™ en Java™ en el que se esperan por las peticiones de los clientes:

```
orb.run()
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
java -cp  jacorb.jar;. EventChannel -ORBInitialPort 3000
                                    -ORBInitialHost localhost
```

3.7.3.2. Canal de Eventos IIOP.Net.

Cuando se va a usar IIOP.Net conviene recordar que este no es una implementación CORBA™ estándar de un ORB, que se trata de un canal Remoting que implementa el protocolo IIOP™ con el que se puede comunicar con los ORBs CORBA™.

Será necesario que el ejecutable que contenga el canal de eventos tenga referencias a las .dlls system.runtime.remoting.dll, *runtime* de .Net Remoting; IIOPChannel.dll, dll de IIOP.Net y EventChannelImpl.dll. EventChannelImpl.dll contiene la implementación del canal de eventos realizado para este proyecto.

Hemos de publicar con .Net Remoting un objeto del tipo *EventChannelImpl* del espacio de nombres *es.uc3m.inf.arcos.EventChannel*, clase que implementa un canal de eventos según la especificación del servicio EventChannel de CORBA™ y además registrarlo en el servidor de nombres.

CORBAInit es una clase del canal de IIOP.Net que nos proporciona acceso al servidor de nombres y a otros servicios de inicialización. Obtenemos un objeto de esa clase con el método estático *GetInit*:

```
CORBAInit corbaInit = CORBAInit.GetInit();
```

IIOP.Net es un canal .Net Remoting que como todos los canales .Net Remoting debe de crearse y registrarse:

```
IiopChannel channel = new IiopChannel(0);
ChannelServices.RegisterChannel(channel);
```

Necesitamos tener un objeto *EventChannelImpl* para posteriormente publicarlo con .Net Remoting. Lo creamos simplemente con el constructor:

```
EventChannelImpl eventChannelImpl = new EventChannelImpl();
```

Publicamos el objeto de tipo *EventChannelImpl* como cualquier servidor .Net Remoting:

```
RemotingServices.Marshal(eventChannelImpl, "EventChannel");
```

Obtenemos una referencia al servidor de nombres con el método *resolveNameService* de la clase *ORBHelper* perteneciente al espacio de nombres *ca.eyt.CORBA*:

```
NamingContext nameService =
      ORBHelper.resolveNameserviceReference(corbaInit, args);
```

Con el método *rebind* de la referencia al servidor de nombres asociamos un nombre a la referencia del canal de eventos:

```
NameComponent[] path =
new NameComponent[] {new NameComponent("EventChannel", "")};
nameService.rebind(path, eventChannelImpl);
```

Como .Net Remoting espera por las peticiones en un hilo creado expresamente para ello hemos de implementar algún método para que la aplicación no termine:

```
while (true) Thread.Sleep(int.MaxValue);
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
EventChannel -ORBInitialPort 3000 -ORBInitialHost localhost
```

3.7.3.3. Proveedor de inyección Jacorb.

Un proveedor de inyección según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que emite eventos al canal a petición propia.

Implementar un proveedor de inyección consiste en hacer un cliente CORBA™ que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *SupplierAdmin*. Con un objeto *SupplierAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPushConsumer* con el que podremos emitir eventos.

Comenzamos por inicializar un *ORB*, el *ORB* es el objeto encargado en un cliente de enviar las peticiones sobre objetos a través de la red, identificado cada uno a través de una referencia CORBA™:

```
ORB orb = ORB.init(args, null);
```

A continuación obtendremos una referencia al servidor de nombres CORBA™ con el que buscaremos una referencia al canal de eventos, esta referencia la obtiene el *ORB* a través de los parámetros de la línea de comandos:

```
org.omg.CORBA.Object objRef = orb
                  .resolve_initial_references("NameService");
```

Con el método *narrow* pasamos de una referencia CORBA™ genérica *org.omg.CORBA.Object* a una del tipo concreto que necesitamos:

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos del tipo *org.omg.CORBA.Object* que con el método *narrow* pasamos a una del tipo *EventChannel*:

```
EventChannel eventChannel = EventChannelHelper
                  .narrow(ncRef.resolve_str("EventChannel"));
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *SupplierAdmin*:

```
SupplierAdmin supplierAdmin = eventChannel.for_suppliers();
```

Con la referencia al objeto *SupplierAdmin* obtenemos un *proxy* de consumidor de inyección con el que podremos emitir eventos:

```
ProxyPushConsumer proxyPushConsumer=supplierAdmin
                                    .obtain_push_consumer();
```

Conectamos el *proxy* de consumidor de inyección como pide la especificación del servicio de eventos:

```
proxyPushConsumer.connect_push_supplier(null);
```

Emitimos un evento consistente en un objeto *Any* que creamos con el *ORB*:

```
Any any = orb.create_any();
any.insert_wchar('H');
proxyPushConsumer.push(any);
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
java -cp  jacorb.jar;. -jar PushSupplier -ORBInitialPort 3000
                                    -ORBInitialHost localhost
```

3.7.3.4. Proveedor de inyección IIOP.Net.

Un proveedor de inyección según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que emite eventos al canal a petición propia.

Será necesario que el ejecutable que contenga el proveedor de inyección tenga referencias a las .dlls system.runtime.remoting.dll, *runtime* de .Net Remoting; IIOPChannel.dll, dll de IIOP.Net y EventChannelImpl.dll. EventChannelImpl.dll contiene la implementación del canal de eventos realizado para este proyecto.

Implementar un proveedor de inyección consiste en hacer un cliente IIOP.Net que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *SupplierAdmin*. Con un objeto *SupplierAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPushConsumer* con el que podremos emitir eventos.

CORBAInit es una clase del canal de IIOP.Net que nos proporciona acceso al servidor de nombres y a otros servicios de inicialización. Obtenemos un objeto de esa clase con el método estático *GetInit*:

```
CORBAInit corbaInit = CORBAInit.GetInit();
```

IIOP.Net es un canal .Net Remoting que como todos los canales .Net Remoting debe de crearse y registrarse. Aquí solamente se necesita la parte cliente del canal:

```
IiopClientChannel clientChannel = new IiopClientChannel();
ChannelServices.RegisterChannel(clientChannel);
```

Obtenemos una referencia al servidor de nombres con el método *resolveNameService* de la clase *ORBHelper* perteneciente al espacio de nombres *ca.eyt.CORBA*:

```
NamingContext nameService =
      ORBHelper.resolveNameserviceReference(corbaInit, args);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos:

```
NameComponent[] name = new NameComponent[]
                        {new NameComponent("EventChannel", "")};
EventChannel eventChannel =
                        (EventChannel) nameService.resolve(name);
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *SupplierAdmin*:

```
SupplierAdmin supplierAdmin = eventChannel.for_suppliers();
```

Con la referencia al objeto *SupplierAdmin* obtenemos un *proxy* de consumidor de inyección con el que podremos emitir eventos:

```
ProxyPushConsumer proxyPushConsumer=supplierAdmin
                                    .obtain_push_consumer();
```

Conectamos el *proxy* de consumidor de inyección como pide la especificación del servicio de eventos:

```
proxyPushConsumer.connect_push_supplier(null);
```

Emitimos un evento con el método *push*, el tipo *Any* del parámetro de este método en OMG™ IDL se mapea a *Object* en C#:

```
proxyPushConsumer.push('H');
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
PushSupplier -ORBInitialPort 3000 -ORBInitialHost localhost
```

3.7.3.5. Consumidor de inyección Jacorb.

Un consumidor de inyección según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que recibe eventos del canal de forma asíncrona a través de una retro llamada.

Implementar un consumidor de inyección consiste en hacer un cliente CORBA™ que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *ConsumerAdmin*. Con un objeto *ConsumerAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPushSupplier* al que conectaremos un objeto *PushConsumer* publicado en el cliente.

Comenzamos por inicializar un *ORB*, el *ORB* es el objeto encargado de enviar peticiones sobre objetos hospedados en otros *ORB*s y de recibir las peticiones sobre objetos de los hospedados por él. Cuando se recibe una petición sobre un objeto el *ORB* se la pasa al *POA* que alberga al objeto sirviente correspondiente que implementa la funcionalidad:

```
ORB orb = ORB.init(args, null);
```

POA es una tipo de adaptador de objeto, en CORBA™ los adaptadores de objeto son los objetos encargados de pasar una petición que les transmite un *ORB* a un objeto sirviente. Para trabajar con *POA*s hay que obtener siempre la referencia al *POA* raíz, que es el gestionado por el *ORB*:

```
POA rootpoa = (POA) orb.resolve_initial_references("RootPOA");
```

Un objeto *POAManager* está asociado con uno o más *POA*s. Es el objeto encargado de controlar el estado de procesamiento de los *POA*s. Cuando se crea un *POAManager* este se encuentra en estado *Holding*, estado en que todas las peticiones entrantes son encoladas, para que el *POA* procese las peticiones es necesario pasarlo al estado *Active* con el siguiente código:

```
rootpoa.the_POAManager().activate();
```

A continuación obtendremos una referencia al servidor de nombres CORBA™ con el que buscaremos una referencia al canal de eventos, esta referencia la obtiene el *ORB* a través de los parámetros de la línea de comandos:

```
org.omg.CORBA.Object objRef = orb
                  .resolve_initial_references("NameService");
```

Con el método *narrow* pasamos de una referencia CORBA™ genérica *org.omg.CORBA.Object* a una del tipo concreto que necesitamos:

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos del tipo *org.omg.CORBA.Object* que con el método *narrow* pasamos a una del tipo *EventChannel*:

```
EventChannel eventChannel = EventChannelHelper
                  .narrow(ncRef.resolve_str("EventChannel"));
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *ConsumerAdmin*:

```
ConsumerAdmin consumerAdmin = eventChannel.for_consumers();
```

Con la referencia al objeto *ConsumerAdmin* obtenemos un proxy de proveedor de inyección:

```
ProxyPushSupplier proxyPushSupplier =
                           consumerAdmin.obtain_push_supplier();
```

POA es una tipo de adaptador de objeto, en CORBA™ los adaptadores de objeto son los objetos encargados de pasar una petición que les transmite un *ORB* a un objeto sirviente, aquí este objeto será el que implemente la retro llamada. Para trabajar con *POA*s hay que obtener siempre la referencia al *POA* raíz, que es el gestionado por el *ORB*:

```
POA rootpoa = (POA) orb.resolve_initial_references("RootPOA");
```

Un objeto *POAManager* está asociado con uno o más *POA*s. Es el objeto encargado de controlar el estado de procesamiento de los *POA*s. Cuando se crea un *POAManager* este se encuentra en estado *Holding*, estado en que todas las peticiones entrantes son encoladas, para que el *POA* procese las peticiones es necesario pasarlo al estado *Active* con el siguiente código:

```
rootpoa.the_POAManager().activate();
```

Creamos un objeto de la clase *pushConsumerPOATie*, que es el sirviente que atiende las retro llamadas. En el constructor de *pushConsumerPOATie* es necesario pasarle un objeto que implemente la interfaz *pushConsumer*, los métodos de este objeto implementaran el comportamiento del cliente ante las retro llamadas:

```
PushConsumerPOATie pushConsumerPOATie =
      new PushConsumerPOATie(new PushConsumerOperations() {
            public void push(Any arg0) throws Disconnected {
                  System.out.println(arg0.extract_wchar());
            }

            public void disconnect_push_consumer() {
                  System.exit(0);
            }
});
```

Hemos de indicarle el objeto *ORB* que hospeda a este objeto:

```
pushConsumerPOATie._this_object(orb);
```

Servant_to_reference asocia un determinado objeto sirviente a una referencia, obtenemos la referencia *org.omg.CORBA.Object* genérica y con *narrow* la convertimos a una de tipo *PushConsumer*, consumidor de inyección:

```
org.omg.CORBA.Object ref = rootpoa
                  .servant_to_reference(pushConsumerPOATie);
PushConsumer pushConsumer = PushConsumerHelper.narrow(ref);
```

Conectamos el *proxy* de consumidor de inyección como pide la especificación del servicio de eventos:

```
proxyPushSupplier.connect_push_consumer(pushConsumer);
```

Por ultimo llamamos al método *run*, bucle principal de un programa servidor CORBA™ en Java™ en el que se esperan por las peticiones de los clientes:

```
orb.run()
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
java -cp  jacorb.jar;. -jar PushConsumer -ORBInitialPort 3000
                                          -ORBInitialHost localhost
```

3.7.3.6. Consumidor de inyección IIOP.Net.

Un consumidor de inyección según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que recibe eventos del canal de forma asíncrona a través de una retro llamada.

Implementar un consumidor de inyección consiste en hacer un cliente IIOP.Net que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *ConsumerAdmin*. Con un objeto *ConsumerAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPushSupplier* al que conectaremos un objeto *PushConsumer* publicado en el cliente.

CORBAInit es una clase del canal de IIOP.Net que nos proporciona acceso al servidor de nombres y a otros servicios de inicialización. Obtenemos un objeto de esa clase con el método estático *GetInit*:

```
CORBAInit corbaInit = CORBAInit.GetInit();
```

Como todos los canales .Net Remoting, IIOP.Net debe de crearse y registrarse:

```
IiopChannel channel = new IiopChannel(0);
ChannelServices.RegisterChannel(channel);
```

Obtenemos una referencia al servidor de nombres con el método *resolveNameService* de la clase *ORBHelper* perteneciente al espacio de nombres *ca.eyt.CORBA*:

```
NamingContext nameService =
      ORBHelper.resolveNameserviceReference(corbaInit, args);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos:

```
NameComponent[] name = new NameComponent[]
                        {new NameComponent("EventChannel", "")};
EventChannel eventChannel =
                        (EventChannel) nameService.resolve(name);
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *ConsumerAdmin*:

```
ConsumerAdmin consumerAdmin = canalDeEventos.for_consumers();
```

Con la referencia al objeto *ConsumerAdmin* obtenemos un *proxy* de proveedor de inyección con el que podremos emitir eventos:

```
ProxyPushSupplier proxyPushSupplier =
                        consumerAdmin.obtain_push_supplier();
```

Creamos un objeto de la clase *pushConsumerPOATie*, que es el sirviente que atiende las retro llamadas:

```
PushConsumerPOATie pushConsumerPOATie =
                                        new PushConsumerPOATie();
```

Añadimos un delegado al evento *onPush* para recibir los eventos del canal:

```
public static void push(object data)
{
      Console.WriteLine(data);
      Environment.Exit(0);
}
pushConsumerPOATie.onPush += new pushHandler(push);
```

Conectamos el proxy de proveedor de inyección al objeto *PushConsumerPOATie* como pide la especificación del servicio de eventos:

```
proxyPushSupplier.connect_push_consumer(pushConsumerPOATie);
```

Como .Net Remoting espera por las peticiones en un hilo creado expresamente para ello hemos de implementar algún método para que la aplicación no termine:

```
while (true) Thread.Sleep(int.MaxValue);
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
PushConsumer -ORBInitialPort 3000 -ORBInitialHost localhost
```

3.7.3.7. Proveedor de extracción Jacorb.

Un proveedor de extracción según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos al que el canal pregunta a través de una retro llamada para que le suministre eventos.

Implementar un consumidor de extracción consiste en hacer un cliente CORBA™ que busque a través de un servidor de nombres de

CORBA™ una referencia a un canal de eventos, le pida a este un objeto *SupplierAdmin*. Con un objeto *SupplierAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPullConsumer* al que conectaremos un objeto *PullSupplier* publicado en el cliente.

Comenzamos por inicializar un *ORB*, el *ORB* es el objeto encargado de enviar peticiones sobre objetos hospedados en otros *ORB*s y de recibir las peticiones sobre objetos de los hospedados por él. Cuando se recibe una petición sobre un objeto el *ORB* se la pasa al *POA* que alberga al objeto sirviente correspondiente que implementa la funcionalidad:

```
ORB orb = ORB.init(args, null);
```

POA es una tipo de adaptador de objeto, en CORBA™ los adaptadores de objeto son los objetos encargados de pasar una petición que les transmite un *ORB* a un objeto sirviente. Para trabajar con *POA*s hay que obtener siempre la referencia al *POA* raíz, que es el gestionado por el *ORB*:

```
POA rootpoa = (POA) orb.resolve_initial_references("RootPOA");
```

Un objeto *POAManager* está asociado con uno o más *POA*s. Es el objeto encargado de controlar el estado de procesamiento de los *POA*s. Cuando se crea un *POAManager* este se encuentra en estado *Holding*, estado en que todas las peticiones entrantes son encoladas, para que el *POA* procese las peticiones es necesario pasarlo al estado *Active* con el siguiente código:

```
rootpoa.the_POAManager().activate();
```

A continuación obtendremos una referencia al servidor de nombres CORBA™ con el que buscaremos una referencia al canal de eventos, esta referencia la obtiene el *ORB* a través de los parámetros de la línea de comandos:

```
org.omg.CORBA.Object objRef = orb
                  .resolve_initial_references("NameService");
```

Con el método *narrow* pasamos de una referencia CORBA™ genérica *org.omg.CORBA.Object* a una del tipo concreto que necesitamos:

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos del tipo *org.omg.CORBA.Object* que con el método *narrow* pasamos a una del tipo *EventChannel*:

```
EventChannel eventChannel = EventChannelHelper
                  .narrow(ncRef.resolve_str("EventChannel"));
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *SupplierAdmin*:

```
SupplierAdmin supplierAdmin = eventChannel.for_suppliers();
```

Con la referencia al objeto *SupplierAdmin* obtenemos un *proxy* de consumidor de inyección con el que podremos emitir eventos:

```
ProxyPullConsumer proxyPullConsumer=supplierAdmin
                                    .obtain_pull_consumer();
```

Creamos un objeto de la clase *pullSupplierPOATie*, que es el sirviente que atiende las retro llamadas. En el constructor de *pullSupplierPOATie* es necesario pasarle un objeto que implemente la interfaz *pullSupplierOperations*, los métodos de este objeto implementaran el comportamiento del proveedor ante las retro llamadas:

```
final PullSupplierPOATie pullSupplierPOATie =
            new PullSupplierPOATie(new PullSupplierOperations() {
                  volatile boolean eventPull=false;
                  public void disconnect_pull_supplier() {
                        System.out.println("Bye.");
                  }

                  public Any try_pull(BooleanHolder has_event)
                                          throws Disconnected {
                        if(this.eventPull) System.exit(0);
                        has_event.value=true;
                        Any any = orb.create_any();
                        any.insert_wchar('H');
                        this.eventPull=true;
                        return any;
                  }

                  public Any pull() throws Disconnected {
                        if(this.eventPull) System.exit(0);
                        Any any = orb.create_any();
                        any.insert_wchar('H');
                        this.eventPull=true;
                        return any;
                  }
});
```

Hemos de indicarle el objeto *ORB* que hospeda a este objeto:

```
pullSupplierPOATie._this_object(orb);
```

Servant_to_reference asocia un determinado objeto sirviente a una referencia, obtenemos la referencia *org.omg.CORBA.Object* genérica y con *narrow* la convertimos a una de tipo *PullSupplier*, proveedor de extracción:

```
org.omg.CORBA.Object ref = rootpoa
                  .servant_to_reference(pullSupplierPOATie);
PullSupplier pullSupplier = PullSupplierHelper.narrow(ref);
```

Conectamos el *proxy* de proveedor de extracción como pide la especificación del servicio de eventos:

```
proxyPullConsumer.connect_pull_supplier(pullSupplier);
```

Por ultimo llamamos al método *run*, bucle principal de un programa servidor CORBA™ en Java™ en el que se esperan por las peticiones de los clientes:

```
orb.run()
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
java -cp  jacorb.jar;. -jar PullSupplier -ORBInitialPort 3000
                                          -ORBInitialHost localhost
```

3.7.3.8. Proveedor de extracción IIOP.Net.

Un proveedor de extracción según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos al que el canal pregunta a través de una retro llamada para que le suministre eventos.

Implementar un consumidor de inyección consiste en hacer un cliente IIOP.Net que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *SupplierAdmin*. Con un objeto *SupplierAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPullConsumer* al que conectaremos un objeto *PullSupplier* publicado en el cliente.

CORBAInit es una clase del canal de IIOP.Net que nos proporciona acceso al servidor de nombres y a otros servicios de inicialización. Obtenemos un objeto de esa clase con el método estático *GetInit*:

```
CORBAInit corbaInit = CORBAInit.GetInit();
```

Como todos los canales .Net Remoting, IIOP.Net debe de crearse y registrarse:

```
IiopChannel channel = new IiopChannel(0);
ChannelServices.RegisterChannel(channel);
```

Obtenemos una referencia al servidor de nombres con el método *resolveNameService* de la clase *ORBHelper* perteneciente al espacio de nombres *ca.eyt.CORBA*:

```
NamingContext nameService =
      ORBHelper.resolveNameserviceReference(corbaInit, args);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos:

```
NameComponent[] name = new NameComponent[]
                        {new NameComponent("EventChannel", "")};
EventChannel eventChannel =
                        (EventChannel) nameService.resolve(name);
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *SupplierAdmin*:

```
SupplierAdmin supplierAdmin = eventChannel.for_suppliers();
```

Con la referencia al objeto *SupplierAdmin* obtenemos un *proxy* de consumidor de inyección con el que podremos emitir eventos:

```
ProxyPullConsumer proxyPullConsumer=supplierAdmin
                                    .obtain_pull_consumer();
```

Creamos un objeto de la clase *pullSupplierPOATie*, que es el sirviente que atiende las retro llamadas:

```
PullSupplierPOATie pullSupplierPOATie =
                                    new PullSupplierPOATie();
```

Añadimos delegados a los eventos encargados de responder al canal ante las peticiones de eventos:

```
static bool eventPull=false;
public static object pull()
{
      if(eventPull) Environment.Exit(0);
      eventPull=true;
      return 'H';
}
public static object try_pull(out bool has_event)
{
      if(eventPull) Environment.Exit(0);
      eventPull=true;
      has_event = true;
      return 'H';
}

pullSupplierPOATie.onTry_pull+=new try_pullHandler(try_pull);
pullSupplierPOATie.onPull += new pullHandler(pull);
```

Conectamos el *proxy* de proveedor de extracción como pide la especificación del servicio de eventos:

```
proxyPullConsumer.connect_pull_supplier(pullSupplierPOATie);
```

Como .Net Remoting espera por las peticiones en un hilo creado expresamente para ello hemos de implementar algún método para que la aplicación no termine:

```
while (true) Thread.Sleep(int.MaxValue);
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
PullSupplier -ORBInitialPort 3000 -ORBInitialHost localhost
```

3.7.3.9. Consumidor de extracción Jacorb.

Un consumidor de extracción según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que recibe eventos del canal de forma *síncrona*, es decir, solicitándoselos al canal.

Implementar un consumidor de extracción consiste en hacer un cliente CORBA™ que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *ConsumerAdmin*. Con un objeto *ConsumerAdmin* para implementar el

proveedor de inyección solicitaremos un objeto *ProxyPullSupplier* con el que podremos recibir eventos.

Comenzamos por inicializar un *ORB*, el *ORB* es el objeto encargado de enviar peticiones sobre objetos hospedados en otros *ORB*s y de recibir las peticiones sobre objetos de los hospedados por él. Cuando se recibe una petición sobre un objeto el *ORB* se la pasa al *POA* que alberga al objeto sirviente correspondiente que implementa la funcionalidad:

```
ORB orb = ORB.init(args, null);
```

A continuación obtendremos una referencia al servidor de nombres CORBA™ con el que buscaremos una referencia al canal de eventos, esta referencia la obtiene el ORB a través de los parámetros de la línea de comandos:

```
org.omg.CORBA.Object objRef = orb
                  .resolve_initial_references("NameService");
```

Con el método *narrow* pasamos de una referencia CORBA™ genérica *org.omg.CORBA.Object* a una del tipo concreto que necesitamos:

```
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos del tipo *org.omg.CORBA.Object* que con el método *narrow* pasamos a una del tipo *EventChannel*:

```
EventChannel eventChannel = EventChannelHelper
                  .narrow(ncRef.resolve_str("EventChannel"));
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *ConsumerAdmin*:

```
ConsumerAdmin consumerAdmin = eventChannel.for_consumers();
```

Con la referencia al objeto *ConsumerAdmin* obtenemos un *proxy* de proveedor de extracción:

```
ProxyPushSupplier proxyPushSupplier =
                        consumerAdmin.obtain_pull_supplier();
```

Conectamos el *proxy* de consumidor de extracción como pide la especificación del servicio de eventos:

```
proxyPushSupplier.connect_pull_consumer(null);
```

Solicitamos un evento al canal con el método *bloqueante pull* que esperará hasta que el canal tenga un evento para devolver. También existe un método no *bloqueante try_pull* que retorna un evento solamente si el canal dispone de uno en ese momentos:

```
System.out.println(proxyPullSupplier.pull().extract_wchar());
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
java -cp  jacorb.jar:. -jar PullConsumer -ORBInitialPort 3000
                                          -ORBInitialHost localhost
```

3.7.3.10. Consumidor de extracción IIOP.Net.

Un consumidor de extracción según la especificación del servicio EventChannel de CORBA™ es un cliente del servidor de canal de eventos que recibe eventos del canal de forma *síncrona*, es decir, solicitándoselos al canal.

Implementar un consumidor de extracción consiste en hacer un cliente IIOP.Net que busque a través de un servidor de nombres de CORBA™ una referencia a un canal de eventos, le pida a este un objeto *ConsumerAdmin*. Con un objeto *ConsumerAdmin* para implementar el proveedor de inyección solicitaremos un objeto *ProxyPullSupplier* con el que podremos recibir eventos.

CORBAInit es una clase del canal de IIOP.Net que nos proporciona acceso al servidor de nombres y a otros servicios de inicialización. Obtenemos un objeto de esa clase con el método estático *GetInit*:

```
CORBAInit corbaInit = CORBAInit.GetInit();
```

IIOP.Net es un canal .Net Remoting que como todos los canales .Net Remoting debe de crearse y registrarse. Aquí solamente se necesita la parte cliente del canal:

```
IiopClientChannel clientChannel = new IiopClientChannel();
ChannelServices.RegisterChannel(clientChannel);
```

Obtenemos una referencia al servidor de nombres con el método *resolveNameService* de la clase *ORBHelper* perteneciente al espacio de nombres *ca.eyt.CORBA*:

```
NamingContext nameService =
      ORBHelper.resolveNameserviceReference(corbaInit, args);
```

Con la referencia al servidor de nombres obtendremos una referencia al canal de eventos:

```
NameComponent[] name = new NameComponent[]
                        {new NameComponent("EventChannel", "")};
EventChannel eventChannel =
                        (EventChannel) nameService.resolve(name);
```

Con la referencia al canal de eventos obtenemos otra a un objeto del tipo *ConsumerAdmin*:

```
ConsumerAdmin consumerAdmin = canalDeEventos.for_consumers();
```

Con la referencia al objeto *ConsumerAdmin* obtenemos un *proxy* de proveedor de extracción con el que podremos emitir eventos:

```
ProxyPullSupplier proxyPullSupplier =
                            consumerAdmin.obtain_pull_supplier();
```

Conectamos el *proxy* de consumidor de extracción como pide la especificación del servicio de eventos:

```
proxyPushSupplier.connect_pull_consumer(null);
```

Solicitamos un evento al canal con el método *bloqueante pull* que esperará hasta que el canal tenga un evento para devolver. También existe un método no *bloqueante try_pull* que retorna un evento solamente si el canal dispone de uno en ese momentos:

```
Console.WriteLine(proxyPullSupplier.pull());
```

Para arrancar este programa con un servicio de nombres en el puerto 3000 de la misma máquina usaremos el siguiente comando:

```
PullConsumer -ORBInitialPort 3000 -ORBInitialHost localhost
```

3.7.4. ActiveMQ.

ActiveMQ es un agente de mensajería y proveedor de JMS 1.1. distribuido bajo la licencia Apache 2.0. ActiveMQ es el proveedor de JMS elegido por Apache para su servidor J2EE™ Geronimo.

JMS es la única API de mensajería soportada por J2EE™. Para que un manejador de mensajes pueda ser proveedor JMS debe de soportar los modelos de mensajería publicación/suscripción y punto a punto. JMS para soportar estos modelos define los conceptos de Queue y Topic:

- Queue es el tipo de destino o canal virtual por el que tenemos que enviar un mensaje para seguir el modelo punto a punto. Solamente un receptor de los suscritos a ese canal recibirá el mensaje.
- Topic es el tipo de destino o canal virtual por el que tenemos que enviar el mensaje para seguir el modelo publicación/suscripción. Todos los clientes suscritos a ese canal recibirán el mensaje.

JMS es un API Java™ que describe la manera en que se han de enviar los mensajes, pero no el protocolo de comunicación o mecanismo usado por el manejador de mensajería. Que un agente de mensajería sea proveedor JMS implica que existen clientes Java™ que soportan el API JMS.

Como se ha visto en otros apartados de este capítulo es posible usar APIs Java™ desde Mono o .Net y enviar así mensajes a través de proveedores JMS. En el caso de JMS este tipo envoltorios deberá ser muy extenso ya que además de ser el propio JMS un API muy amplia no está aislada, requiere del API JNDI™.

ActiveMQ además de ser proveedor de JMS incorpora servidores de distintos protocolos a los que llama conectores a través de los cuales también podemos enviar y recibir mensajes.

STOMP (Streaming Text Orientated Messaging Protocol) es un protocolo de mensajería diseñado con el fin de poder escribir clientes de manera sencilla en cualquier lenguaje con soporte de sockets. La implementación oficial del servidor STOMP es un conector de ActiveMQ.

Además del servidor STOMP incluido en ActiveMQ existe una implementación completa de STOMP, tanto de servidor como de cliente, en Java™ bajo licencia LGPL llamada SerSTOMP.

Usando como modelo la versión 0.4.0 de SerSTOMP y con la licencia LGPL se ha desarrollado como parte de este proyecto otra implementación completa del protocolo en C# que hemos llamado #STOMP.

Los detalles de la implementación de #STOMP aparecen en el apartado 4.2.1.de esta documentación.

3.7.4.1. ActiveMQ como servidor STOMP.

La versión de ActiveMQ que hemos usado para probar este ejemplo es la 3.1. :

Aunque ActiveMQ puede arrancarse como servidor empotrado dentro de otra aplicación Java™ también incorpora dos *scripts*, uno para Windows y otro para la *shell sh* de Unix®/Linux®, con el que podemos arrancar el servidor.

El comando para arrancar el servidor con la configuración estándar del mismo es:

ActiveMQ

Con la configuración estándar no tenemos activado el conector STOMP, por lo que hemos de realizar un sencillo archivo de configuración. Los archivos de configuración son ficheros XML que siguen la DTD activemq.dtd. En los archivos de configuración especificamos los conectores y los mecanismos de persistencia de mensajes.

Para que nuestro servidor arranque el conector STOMP añadiremos las siguientes líneas en la zona dónde se especifican el resto de conectores:

```
<connector>
      <serverTransport uri="stomp://localhost:61626"/>
</connector>
```

Un documento de completo para configurar el servidor ActiveMQ con conectores STOMP, TCP (para poder usarlo con clientes JMS) y sin persistencia es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//ACTIVEMQ//DTD//EN"
                        "http://activemq.org/dtd/activemq.dtd">
<beans>
  <broker>
    <connector>
      <tcpServerTransport uri="tcp://localhost:61616"
      backlog="1000" useAsyncSend="true"
      maxOutstandingMessages="50"/>
    </connector>
    <connector>
      <serverTransport uri="stomp://localhost:61626"/>
```

```
    </connector>
    <persistence>
      <vmPersistence/>
    </persistence>
  </broker>
</beans>
```

El comando para arrancar el servidor con esta configuración es:

ActiveMQ configuración.xml

STOMP simplemente incorpora el concepto de destino para indicar hacia dónde se transmiten los mensajes. Para STOMP un destinto es una cadena de caracteres que indica a donde y desde donde recibimos mensajes.

Como ya hemos explicado antes ActiveMQ es un proveedor JMS que está obligado a soportar los conceptos Queue y Topic de JMS. El servidor STOMP de ActiveMQ especifica que el identificador STOMP de cada q*ueue* JMS es el nombre de esa cola con el prefijo */queue/*, de igual manera también específica que el identificador STOMP de cada *topic* JMS es el nombre de dicho *topic* con el prefijo */topic/*.

El servidor STOMP de ActiveMQ no transmite ningún mensaje cuyo destino no comienze por */queue/* o */topic/*. En el caso de los servidores de SerSTOMP y #STOMP estos prefijos carecen de valor y los mensajes son transmitidos siempre a todos los clientes que se encuentren suscritos a dichos destinos.

3.7.4.2. SerSTOMP.

La versión de STOMP que hemos usado para la realización del proyecto es la 0.4.0. Para usar esta versión de SerSTOMP hemos de añadir al *classpath* el jar serstomp-0.4.0.jar.

Aunque STOMP como se explicará en el apartado 4.2.1.2. incorpora operaciones para enviar con acuse de recibo aquí vamos a comentar solamente los pasos necesarios para leer y escribir *strings* con Java™.

SerSTOMP incorpora una implementación de servidor. Para crear un servidor con SerSTOMP basta con crear un objeto de la clase *Server* indicando el puerto TCP dónde queremos que escuche:

```
Server server = new Server( 61626 );
```

Podemos crear clientes del servidor con los cuales este se comunica a través de estructuras de memoria dentro de la misma máquina virtual:

```
Stomp client=server.getClient();
```

STOMP es un protocolo en el que los clientes se comunican a través de sockets. Si queremos un cliente que se comunique con el servidor a través de TCP lo podemos hacer también creando un objeto de la clase *Client* al que le indicamos el nombre o la dirección IP de la máquina, el

puerto TCP dónde escucha el servidor, el identificador de usuario y el *password* de usuario:

```
Client client = new Client( "localhost", 61626, "ser", "ser" );
```

Los identificadores y los *password* de usuario no se usan por parte del servidor SerSTOMP, pero aparecen en el protocolo.

Una vez que tenemos un cliente podemos suscribirnos a destinos y enviar mensajes. Para suscribirnos usaremos el método *subscribe*, al que le pasaremos el nombre del destino y un objeto que implemente la interfaz *Listener* para recibir los eventos:

```
client.subscribe("/topic/helloworld",new Listener() {
      public void message(Map headers, String body) {
            System.out.println(body);
}});
```

Para enviar mensajes a un destino tenemos el método *send*:

```
client.send("/topic/helloworld","HelloWorld");
```

3.7.4.3. #STOMP.

#STOMP es la implementación hecha dentro del proyecto en C# basada en SerSTOMP. Para usarla en un ejecutable hemos de añadir como referencia la .dll SharpStomp.dll.

Aunque STOMP como se explicará en el apartado 4.2.1.2. incorpora operaciones para enviar con acuse de recibo aquí vamos a comentar solamente los pasos necesarios para leer y escribir *strings* con C#.

#STOMP incorpora una implementación de servidor. Para crear un servidor con SerSTOMP basta con crear un objeto de la clase *Server* indicando el puerto TCP dónde queremos que escuche:

```
Server server = new Server( 61626 );
```

Podemos crear clientes del servidor con los cuales este se comunica a través de estructuras de memoria dentro de la misma máquina virtual:

```
Stomp client=server.Client;
```

STOMP es un protocolo en el que los clientes se comunican a través de sockets. Si queremos un cliente que se comunique con el servidor a través de TCP lo podemos hacer también creando un objeto de la clase *Client* al que le indicamos el nombre o la dirección IP de la máquina, el puerto TCP dónde escucha el servidor, el identificador de usuario y el *password* de usuario:

```
Client client = new Client( "localhost", 61626, "ser", "ser" );
```

Los identificadores y los *password* de usuario no se usan por parte del servidor #STOMP, pero aparecen en el protocolo.

Una vez que tenemos un cliente podemos suscribirnos a destinos y enviar mensajes.

Para suscribirnos usaremos el método *subscribe*, al que le pasaremos el nombre del destino y un delegado *Listener* para recibir eventos:

```
client.subscribe("/topic/helloworld",new Listener(message));


public static void message(IDictionary headers, string body)
{
      Console.WriteLine(body);
}
```

Para enviar mensajes a un destino disponemos del método *send*:

```
client.send("/topic/helloworld","HelloWorld");
```

3.7.5. XMLBlaster.

XMLBlaster es un middleware orientado a mensajes que soporta los modelos publicación/suscripción y punto a punto. Los mensajes se describen a través de meta información codificada en XML.

El servidor XMLBlaster se arranca con el siguiente comando:

```
java -jar xmlblaster.jar
```

Los clientes para comunicarse con el servidor pueden usar CORBA™, RMI y XMLRPC. Además XMLBlaster incorpora un interfaz gráfico con el que podemos auditar el servidor.

El API definido mediante OMG™ IDL de XMLBlaster contiene relativamente pocas operaciones, aunque debemos tener en cuenta que muchos parámetros son *string* que contienen documentos XML, especificados según los DTD incluidos en la distribución.

Cada mensaje de XMLBlaster está compuesto por una cabecera XML, un cuerpo y un *string* XML que llama *QOS*. El *string QOS* (Quality Of Service) es el que informa a XMLBlaster de cómo debe tratar el mensaje.

Los clientes se subscriben mediante un *string* XML en el que indican la cabecera XML literal de los mensajes que quieren recibir o un camino XPath que se pueda encontrar en el XML de dicha cabecera.

El XML de la cabecera también sigue el DTD key.dtd que trae la distribución de XMLBlaster. Este XML obliga a que en todas las cabeceras esté el *tag key* como raíz y que este contenga el parámetro *oid*. El parámetro *oid* es el identificador de tópico.

Para suscribirnos a todos los mensajes que contiene el servidor y que seamos avisados siempre que ocurran cambios el *string* XML que le hemos de pasar al método de suscripción es:

```
<key oid='' queryType='XPATH'>/xmlBlaster/key</key>
```

Si quisiéramos suscribirnos solamente al mensaje cuyo *oid* es 10 pasaríamos:

```
<key oid='' queryType='XPATH'>/xmlBlaster/key[@oid='10']</key>
```

Con estos ejemplos no estamos aprovechando las capacidades de XPath que permite suscribirse a varios mensajes a la vez.

Vamos a suponer que para indicar que un grupo de mensajes es de un proveedor este añade un *tag first* en todas sus cabeceras por debajo del *tag key* y nosotros queremos suscribirnos a todos. Pues simplemente construimos el camino XPath y pasamos el siguiente XML:

```
<key oid='' queryType='XPATH'>/xmlBlaster/key/first</key>
```

Cuando llega un mensaje al servidor sin ninguna restricción indicada a través de su *QOS* este será enviado a través de retro llamadas a todos los clientes cuyas suscripciones acepten la cabecera de dicho mensaje.

3.7.5.1. #Blaster.

Para el siguiente ejemplo hemos usado la versión 1.0.7 de SharpBlaster y la librería #Blaster escrita sobre XMLRPC que hemos realizado como parte de este proyecto.

Nuestro programa escrito en C# deberá referencia el .dll SharpBlaster.dll.

Vamos a registrar el cliente con un *callback* para recibir las notificaciones de actualización de mensajes en modo asíncrono. Comenzamos creando un objeto *SharpBlaster* con la dirección del servidor y posteriormente no registramos con el método *login* en el servidor, en el método *login* pasamos el nombre de usuario, su contraseña y el puerto donde abrimos el servidor XMLRPC del *callback*:

```
SharpBlaster.SharpBlaster sharpBlaster = new
      SharpBlaster.SharpBlaster("http://192.168.0.1:8080/RPC2");
sharpBlaster.login("user","password",8081);
```

A continuación vamos a suscribir el cliente a todos los mensajes del servidor. Vamos a usar el método *subscribeXPath* de *SharpBlaster* que nos facilita el uso de XPath:

```
sharp.subscribeXPath("/xmlBlaster/key/first");
```

Con el método *subscribe* podríamos hacer lo mismo de la siguiente manera:

```
sharp.subscribe("<key oid='' queryType='XPATH'
                                    >/xmlBlaster/key</key ");
```

Para recibir los eventos usamos el *event update* de la clase *SharpBlaster*:

```
sharp.update += new update(sharp_update);

private static void sharp_update(string cbSessionId, string
updateKey, byte[] content, string updateQos)
{
      Console.WriteLine(updateKey);
}
```

Para cancelar una suscripción está disponible el método *unsubscribe* y su versión de XPath *unsubscribeXPath*:

```
sharp.unSubscribeXPath("/xmlBlaster/key");
```

Para publicar un nuevo mensaje en el servidor cuyo *oid* sea 50 usamos el método *publish*:

```
sharp.publish("<key oid='50'><first/></key>",
            new UTF8Encoding().GetBytes("Hola Mundo"), "<qos/>");
```

Para borrar el anterior mensaje disponemos del método *eraseXPath*, aunque podríamos también usar el método *erase* de manera análoga a lo que pasa con *subscribeXPath* y *subscribe*:

```
sharp.eraseXPath("/xmlBlaster/key[@oid='50']");
```

#Blaster también permite leer los mensajes de manera *síncrona* con los métodos *get* y *getXPath*:

```
MessageUnit[] msgArr = sharp.
                        getXPath("/xmlBlaster/key[@oid='50']");
```

Por ultimo, para cerrar la conexión con el servidor se dispone del método *logout*:

```
sharp.logout();
```

3.8. Interoperatividad con Enterprise Service Bus.

Un Enterprise Service Bus (ESB) o bus de integración empresarial es un entorno de trabajo de servicios y mensajes escalable y tolerante a fallos que:

- Proporciona un mecanismo transparente de comunicar servicios heterogéneos a través de un variado grupo de protocolos de mensajería.
- Proporciona una capa de mensajería común para que aplicaciones empresariales, servicios y componentes puedan conectarse y comunicarse.
- Puede transmitir mensajes de forma *síncrona* y asíncona entre puntos de servicio o terminales, realizar adaptaciones de mensajes y *securizarlos* de acuerdo a las necesidades de los puntos de servicio.

3.8.1. Introducción al Enterprise Service Bus.

El elemento central de cualquier ESB es un bus de mensajes que es usado como medio de comunicación entre diferentes componentes o aplicaciones. Normalmente se usa JMS como bus de mensajes, aunque puede ser usado cualquier software *middleware* de Mensajes (MOM) como los vistos en el apartado 3.7.

Service Oriented Architecure (SOA) es una arquitectura de aplicaciones en la que todas las funciones o servicios están definitivas en un lenguaje de descripciones y posee interfaces especiales para realizar una acción en el proceso del negocio. Cada interacción es independiente de cualquier otra en el procedimiento, por lo que incluso los protocolos de comunicación de los diferentes elementos, también lo son. De esta forma cualquier elemento, con cualquier sistema operativo y cualquier lenguaje, puede utilizar el servicio.

Un servicio de acuerdo a la definición de SOA es una función sin estado, auto-contenida, que acepta una o varías llamadas y devuelve una o varías respuestas mediante una interfaz bien definida. Los servicios no dependen del estado de otras funciones o procesos.

ESB sigue la arquitectura SOA y es por eso que se define como un entorno de trabajo para servicios. SOA es la arquitectura, como vimos en el apartado 3.6.1. , en la que se basan los servios web. Es por eso que la mayoría de los ESBs incluyen SOAP como protocolo de comunicación con los clientes o terminales.

3.8.2. Mule como puente entre SOAP y ActiveMQ.

Mule es un *framework* para el desarrollo de Enterprise Service Bus. Mule puede trabajar con proveedores de JMS como ActiveMQ y publicar y usar servicios Web a través de Apache Axis.

Apache Axis es un *framework* SOAP que hemos visto en el apartado 3.6.2. de esta documentación.

En el apartado 3.7.3.1 se explica como arrancar ActiveMQ como servidor de clientes JMS y STOMP. En este apartado vamos a configurar Mule como cliente JMS de ese servicio. Cuando se envíen mensajes a través de Mule al servidor ActiveMQ estos también podrán leerse con los clientes del protocolo STOMP.

Para arrancar Mule lo hacemos a través de un *script* al que le pasamos un archivo de configuración XML con la estructura del DTD mule-configuration.dtd:

```
mule -config mule.xml
```

El corazón de Mule es un contenedor de objetos UMOs. Un UMO, Universal Message Object, es un objeto JavaBean™ estándar que puede recibir y enviar información a través de *Endpoints*. Un *Endpoint* es un canal de comunicación entre dos componentes. Los *Endpoints* proporcionan una

manera de permitir hablar a los objetos de una manera unificada a través de varios protocolos.

Un conector es un objeto usado por Mule para crear distintos medios a través de los cuales enviar y recibir información. Entre esto medios podemos encontrar clientes o servidores de distintos protocolos, clientes JMS, ficheros o una consola de texto.

Para registrar un conector que se ocupe de la consola añadimos el siguiente texto en el archivo de comunicación:

```
<connector name="SystemStreamConnector"
classNamе="org.mule.providers.stream.SystemStreamConnector">
  <properties>
      <property name="promptMessage"
                              value="Please enter something: "/>
      <property name="messageDelayTime" value="1000"/>
  </properties>
</connector>
```

Para comunicarnos con el servidor ActiveMQ hemos de incorporar a nuestro *classpath* los jar activemq-3.1.jar y concurrent-1.3.4.jar, además tenemos que especificar el siguiente conector en el archivo XML:

```
<connector name="activeMQConnector"
      className="org.mule.providers.jms.JmsConnector">
   <properties>
            <property name="connectionFactoryJndiName"
                        value="ConnectionFactory"/>
      <property name="jndiInitialFactory"
      value="org.activemq.jndi.ActiveMQInitialContextFactory"/>
      <property name="specification" value="1.1"/>
      <map name="connectionFactoryProperties">
      <property name="brokerURL" value="tcp://localhost:61616"/>
      </map>
   </properties>
</connector>
```

Vamos a añadir ahora un repetidor de mensajes a través de un componente UMO llamado *echo*. Este repetidor pasará los mensajes obtenidos de un servicio web SOAP y de la consola cada vez que se realice un retorno de carro a un *topic* del servidor ActiveMQ:

```
<mule-descriptor name="Echo"
      implementation="org.mule.components.simple.EchoComponent">
    <inbound-router>
      <endpoint address="stream://System.in"/>
      <endpoint address="axis:http://localhost:8081/services"/>
    </inbound-router>
    <outbound-router>
      <router
className="org.mule.routing.outbound.OutboundPassThroughRouter">
      <endpoint address="jms://topic:helloworld"
                  connector="activeMQConnector"/>
      </router>
    </outbound-router>
</mule-descriptor>
```

A continuación con otro repetidor como el anterior escribimos por consola los mensajes enviados desde un *topic* del servidor ActiveMQ:

```
<mule-descriptor name="Receive"
      implementation="org.mule.components.simple.EchoComponent">
      <inbound-router>
            <endpoint address="jms://topic:helloworld"
                              connector="activeMQConnector"/>
      </inbound-router>
      <outbound-router>
         <router
className="org.mule.routing.outbound.OutboundPassThroughRouter">
            <endpoint address="stream://System.out"/>
         </router>
      </outbound-router>
</mule-descriptor>
```

Con la configuración anterior tenemos en la dirección *http://localhost:8081/services* un servicio SOAP a través del cual podremos enviar a través de Mule mensajes al topic de ActiveMQ.

Siguiendo los pasos descritos en el apartado 3.6.2.3 creamos desde el WSDL del servicio web de Mule un proxy en C# implementado en la clase *Proxy0Service*. Dicha clase incorpora un método *echo* para enviar mensajes. Al llamar a dicho método se produce una excepción por el tratamiento del retorno. Como el retorno no se necesita, simplemente se captura la excepción:

```
Proxy0Service proxy0Service=new Proxy0Service();
proxy0Service.Url = "http://localhost:8081/services/Echo";
try
{
      proxy0Service.echo("Hello World");
}
catch(System.InválidoperationException)
{}
```

3.8.3. Apache ServiceMix como puente entre SOAP y ActiveMQ.

Apache ServiceMix es el ESB elegido por Apache para su servidor J2EE™ Geronimo. Esta construido basándose en el estándar JBI de Java™.

La especificación (JSR 208) Java™ Business Integration (JBI) define el núcleo de un bus de integración orientado al servicio y la arquitectura de componentes para SOA. Estandariza la arquitectura común de enrutamiento de mensajes, las interfaces de conexión para motores de servicios y enlaces, y un mecanismo (Descripción de Servicios Compuestos) para combinar distintos servicios en una sola unidad de trabajo ejecutable y auditable.

En el apartado 3.7.3.1 se explica como arrancar el ActiveMQ como servidor de clientes JMS y STOMP. En este apartado vamos a configurar ServiceMix como cliente JMS de ese servicio. Cuando se envíen mensajes a través de ServiceMix al servidor ActiveMQ estos también podrán leerse con los clientes del protocolo STOMP.

ServiceMix es básicamente un contenedor de componentes JBI. *ActivationSpec* es la clase con que ServiceMix registra los componentes JBI en el bus, nosotros vamos a registrar dichos componentes usando un archivo de configuración XML. La estructura básica del archivo de configuración de ServiceMix es:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
      xmlns:spring="http://xbean.org/schemas/spring/1.0"
      xmlns:sm="http://servicemix.org/config/1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=
"http://xbean.org/schemas/spring/1.0 ../../conf/spring-beans.xsd
 http://servicemix.org/config/1.0 ../../conf/servicemix.xsd">

  <!-- El contenedor JBI -->
  <sm:container spring:id="jbi"
            useMBeanServer="true"
            createMBeanServer="true"
            dumpStats="true"
            statsInterval="10">

      <sm:activationSpecs>

            <sm:activationSpec ... >
              <sm:component>
                <bean xmlns="http://xbean.org/schemas/spring/1.0"
                      class="...">
                  <property name="..." value="..."/>
                  ...
                  <property name="..." value="..."/>
                </bean>
              </sm:component>
            </sm:activationSpec>

            ...

            <sm:activationSpec ... >
              <sm:component>
                <bean xmlns="http://xbean.org/schemas/spring/1.0"
                      class="...">
                  <property name="..." value="..."/>
                  ...
                  <property name="..." value="..."/>
                </bean>
              </sm:component>
            </sm:activationSpe

      </sm:activationSpecs>
  </sm:container>

  <!-- Otros componentes -->
  <bean ... >
  ...
  </bean>
  ...
  <bean ... >
  </bean>

</beans>
```

Vamos a utilizar el mismo cliente SOAP del apartado anterior. ServiceMix no tiene ningún componente JBI para actuar como servidor HTTP con soporte para SOAP. El componente servidor HTTP de ServiceMix simplemente recoge las peticiones XML que se le hagan y las transmite sin procesar a otro componente.

Con el siguiente código arrancaremos el servidor HTTP en el puerto 8081, el nombre del servicio es *httpBinding* y las peticiones HTTP serán reenviadas al servicio *transformer*:

```
<sm:activationSpec componentName="httpReceiver"
            service="httpBinding" endpoint="httpReceiver"
                        destinationService="transformer">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="org.servicemix.components.http.HttpConnector">
      <property name="host" value="localhost"/>
      <property name="port" value="8081"/>
    </bean>
  </sm:component>
</sm:activationSpec>
```

Necesitaremos un componente JBI que inyecte eventos en el canal, este componente JMS usará una factoría de conexiones de ActiveMQ que declararemos en la sección destinada a componentes no JBI. Aquí indicaremos la dirección del agente de mensajería ActiveMQ:

```
<bean id="jmsFactory"
              class="org.activemq.pool.PooledConnectionFactory">
  <property name="connectionFactory">
    <bean class="org.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL">
        <value>tcp://localhost:61616</value>
      </property>
    </bean>
  </property>
</bean>
```

Como vimos en el aparatado 3.4.7. en JMS los canales por los que circulan los eventos son *topic* en el modelo publicación/suscripción, también conocido como modelo pub/sub. En el componente de emisión de eventos JMS de ServiceMix un *topic* es un destino del dominio *pubSub*. El servicio de este componente es *outputSender*:

```
<sm:activationSpec componentName="outputSender"
                                service="outputSender">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
        class="org.servicemix.components.jms.JmsSenderComponent">
      <property name="template">
        <bean class="org.springframework.jms.core.JmsTemplate">
          <property name="connectionFactory">
            <ref local="jmsFactory"/>
          </property>
          <property name="defaultDestinationName"
                                          value="helloworld"/>
          <property name="pubSubDomain" value="true"/>
        </bean>
      </property>
     </bean>
```

```
   </sm:component>
</sm:activationSpec>
```

Entre el componente que recibe las entradas HTTP y el componente que envía datos al canal vamos a situar uno que se encargue de transformar el complejo XML de los paquetes SOAP en algo más sencillo. Para hacer esta transformación nos vamos a apoyar en plantillas XSLT.

XSLT o XSL Transformaciones es un estándar de la organización W3C que presenta una forma de transformar documentos XML en otros XMLs e incluso a formatos que no son XML.

Lamentablemente, aunque XSLT si incorpore métodos para transformas XML en texto plano, ServiceMix solamente es capaz de emitir XML con esta tecnología.

Los mensajes XML que enviaremos a través de XML tendrán el siguiente formato:

```
<message>
   Contenido del mensaje.
</message>
```

El servicio del componente encargado de transformar el XML es *transformer*. El XML resultante de la transformación realizada con la plantilla XSLT se enviará al servicio *outputSender* para su envío a través del agente de mensajería ActiveMQ.

El componente XSLT por defecto envía el XML resultante de la plantilla que no se entrega al servicio de mensajería de vuelta como retorno de la petición HTTP realizada al servicio *httpBinding*, para que esto no ocurra se lo indicamos al servicio de XSLT. Naturalmente, el instrumento fundamental para la transformación es la plantilla XSLT, el fichero que contiene dicha plantilla es transform.xsl:

```
<sm:activationSpec componentName="transformer"
  service="transformer" endpoint="transformer"
                              destinationService="outputSender">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="org.servicemix.components.xslt.XsltComponent">
      <property name="xsltResource"
                               value="classpath:transform.xsl"/>
      <property name="disableOutput" value="true"/>
    </bean>
   </sm:component>
</sm:activationSpec>
```

En la plantilla XSLT, como ya habíamos comentado antes, debe de indicarse que XML irá al servicio de envío JMS. ServiceMix incluye unas extensiones de XSLT para tratar con el *enrutado* de mensajes. En este ejemplo *jbi:invoke* es la extensión de ServiceMix encargada de indicar el XML a enviar a través del tópico JMS.

xsl:copy-of es un *tag* que copia al documento de salida el XML las partes del mismos devueltas por una consulta XPath. XPath es un lenguaje (basado en XML) y estandarizado por W3C que permite seleccionar

subconjuntos de un documento XML. En esta plantilla lo que se selecciona es el texto de todas las *tags in0*. *in0* son las *tags* que transportan el parámetro del método SOAP que llamamos en el ejemplo de cliente del apartado anterior:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:jbi="xalan://org.servicemix.components.xslt.
                                                XalanExtension"
  extension-element-prefixes="jbi" version="1.0">
    <xsl:template match="/*">
       <jbi:invoke service="my:outputSender">
         <message>
             <xsl:copy-of select="//in0/text()"/>
         </message>
       </jbi:invoke>
    </xsl:template>
</xsl:stylesheet>
```

El comando para arrancar ServiceMix es:

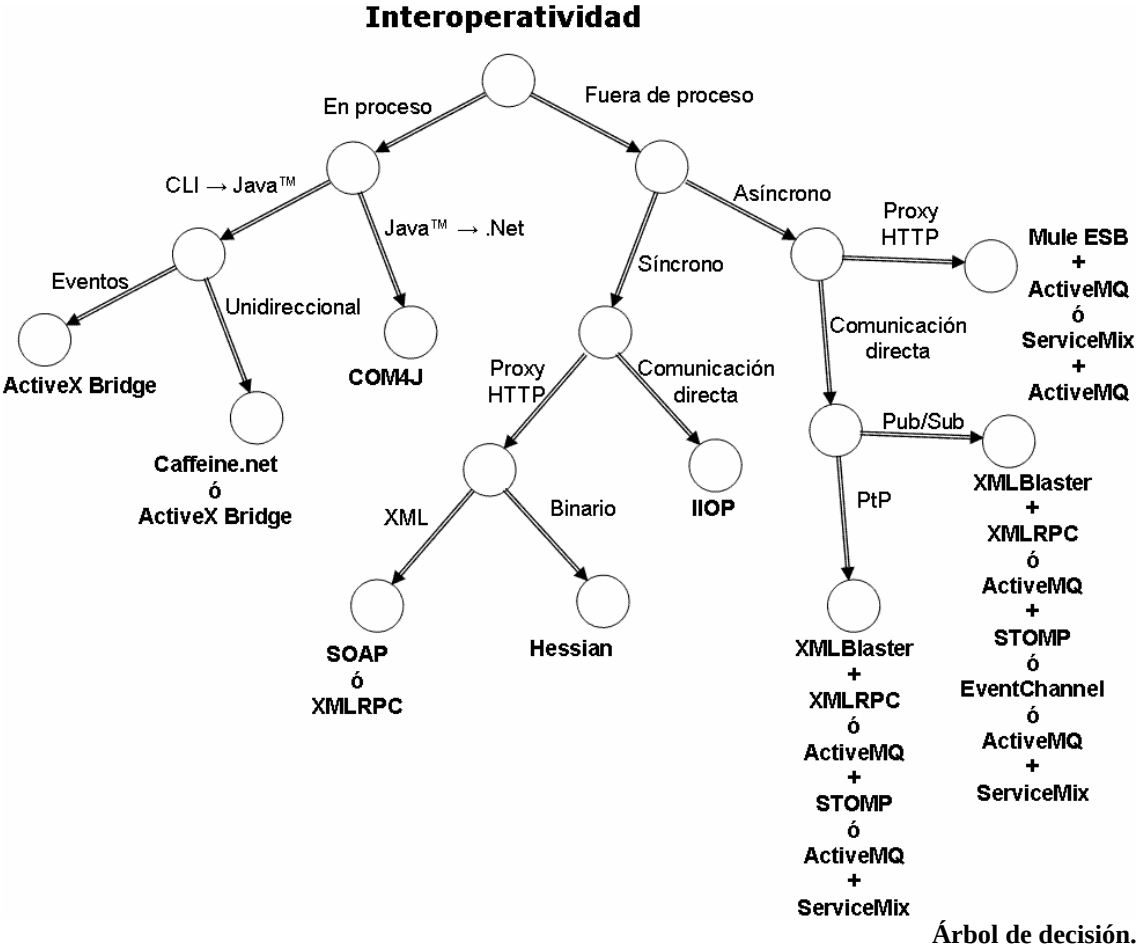
```
Servicemix servicemix.xml
```

3.9. Árbol de decisión.

Se han mencionado un conjunto de soluciones para la interoperatividad entre aplicaciones escritas para las plataformas de desarrollo Java™ y CLI. Hemos visto que cada una trataba el tema de la interoperatividad con el objetivo de comunicar aplicaciones de una manera distinta en función de unas determinadas necesidades.

Conocer una manera de comunicar aplicaciones Java™ y CLI no es suficiente para solucionar todos los problemas en los que se necesite interoperatividad entre estas dos plataformas de desarrollo. El siguiente árbol de decisión pretende ofrecer una manera rápida y visual de llegar a la solución más acorde por un problema entre las tratadas en este documento.

La persistencia no ha sido contemplada en el árbol de decisión. Entendemos que XML y las bases de datos pueden usarse para pasar información conjuntamente con alguna de las soluciones del árbol.



Árbol de decisión.

Los conceptos "en proceso" y "fuera de proceso" se han tomado de COM. Un servidor "en proceso" es una librería dinámica, las llamadas a esa librería se realizan dentro del mismo proceso. Un servidor "fuera de proceso" es aquel servidor COM implementado mediante un servicio al que se accede a través de llamadas a procedimientos remotos. Aquí a lo que queremos hace referencia es que las comunicaciones se realizan entre dos procesos distintos.

3.10. Tendencias futuras.

Sun™ ha llegado a un acuerdo con Microsoft® para facilitar la interoperatividad de Java con el *framework* de comunicaciones orientado al servicio WCF.

WCF (Windows Communication Foundation), anteriormente llamado Indigo, es un *framework* que unifica las tecnologías de computación distribuida ASMX, .Net Remoting, WSE, Enterprise Services y MSMQ. Aunque Microsoft® se plantea continuar soportando cada una de las tecnologías anteriores se pretende que los nuevos desarrollos usen exclusivamente WCF. De momento el proyecto Mono no plantea implementar WCF. Para usar WCF deberemos tener instalada la versión 2.0 del .Net Framework.

.Net Remoting se ha usado en los apartados 3.5.2 con IIOP,3.6.2 con SOAP y 3.6.3. con XML-RPC. MSMQ ha sido tratado en el apartado 3.7.2.

ASMX es el nombre corto por el que se conocen a los servicios web ASP.NET. ASMX usa la infraestructura para aplicaciones web ASP.NET para el desarrollo de servidores SOAP. Los servidores web que soportan ASP.Net son IIS y Cassini de Microsoft sobre .Net Framework y XSP sobre Mono. Además existen dos módulos para el servidor web Apache que son *mod_aspdotnet* sobre .Net Framework y *mod_mono* para la plataforma Mono. Los servicios ASMX pueden ser consumidos desde las plataformas Java y CLI construyendo clientes de igual manera que los del 3.6.2. En el proyecto hemos preferido realizar los ejemplos de servidor SOAP con .Net Remoting para evitar tener que explicar la configuración de un servidor Web, es el mismo criterio por el que se ha usado Jetty® en lugar de Tomcat para los servidores Java.

Enterprise Services es la infraestructura del .Net Framework para desarrollar servicios COM+. COM+ es una extensión de COM, que tratamos en el apartado 3.1., para la integración de este con MTS, servicio de transacciones de Microsoft. Esta considerado como la respuesta de Microsoft a las EJB™ de Java™. Desde el punto de vista de la interoperatividad entre Java y .Net carece de interés.

WSE es una implementación de varías especificaciones WS*. Entre las distintas especificaciones tenemos por ejemplo WS-Attachments para añadir ficheros a un mensaje SOAP fuera del envoltorio sin tener que serializarlos a XML, WS-Security para *securizar* servicios web a nivel de mensajes y WS-MetadataExchange que define el formato de los mensajes de intercambio de meta datos que describen un servicio web.

Sun™ planea implementar las especificaciones WS* relativas a Mensajería basada en SOAP, Metadata, Seguridad y Calidad de Servicio, y estarán hacerlas disponibles para la comunidad a través del servidor de aplicaciones J2EE5™ de fuente abierta Glassfish.

4. Desarrollo de librerías MOM sobre CLI.

4.1. Desarrollo de un servicio de eventos CORBA™ con IIOP.Net.

J-Integra™ Espresso, Middcor.Net™ y Borland® Janeva™ son implementaciones completas de CORBA™ ORB para el .Net Framework, J-Integra™ Espresso incluso está soportado en Mono. Lamentablemente estas soluciones son comerciales y hay que pagar licencia para poder usarlas.

En la introducción del proyecto se mencionó que solamente se iban a usar soluciones gratuitas y cuyas fuentes estuviesen disponibles en Internet.

Remoting.CORBA e IIOP.Net son dos canales .Net Remoting que implementan el protocolo IIOP™. IIOP™ es un protocolo que obligatoriamente deben implementar los ORB que sigan la especificación CORBA™ 2.0 o cualquiera posterior a esta. Remoting.CORBA e IIOP.Net incorporan funcionalidad suficiente para implementar el servicio de eventos.

La implementación IIOP.Net además del canal incorporar herramientas para generar IDL desde código intermedio y viceversa. Por este motivo esta es la que hemos seleccionando la implementación de este servicio.

4.1.1. El servicio de eventos de CORBA™.

Common Object Request Broker Architecture (CORBA™) es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

CORBA™ es más que una especificación multiplataforma, también define servicios habitualmente necesarios como seguridad y transacciones.

4.1.1.1. Introducción a CORBA™ y a los servicios CORBA™.

CORBA™ fue definido y está controlado por el Object Management Group (OMG) que define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida.

En un sentido general CORBA™ *envuelve* el código escrito en otro lenguaje en un paquete que contiene información adicional sobre las capacidades del código que contiene, y sobre cómo llamar a sus métodos. Los objetos que resultan pueden entonces ser invocados desde otro programa (u objeto CORBA™) desde la red. En este sentido CORBA™ se puede considerar como un formato de documentación legible por la máquina, similar a un archivo de cabeceras pero con más información.

CORBA™ utiliza un lenguaje de definición de interfaces (IDL) para especificar los interfaces con los servicios que los objetos ofrecerán. CORBA™ puede especificar a partir de este IDL la interfaz a un lenguaje

determinado, describiendo cómo los tipos de dato CORBA™ deben ser utilizados en las implementaciones del cliente y del servidor. Implementaciones estándar existen para Ada, C, C++, Smalltalk, Java™ y Python. Hay también implementaciones para Perl y TCL.

Al compilar una interfaz en IDL se genera código para el cliente y el servidor (que implementa el objeto). El código del cliente sirve para poder realizar las llamadas a métodos remotos. Es el conocido como *stub*, el cual incluye un *proxy* (representante) del objeto remoto en el lado del cliente. El código generado para el servidor consiste en unos *skeletons* (esqueletos) que el desarrollador tiene que rellenar para implementar los métodos del objeto.

Para cada servicio del estándar CORBA™ la OMG™ provee una especificación formal descrita en OMG™ IDL (como invocar cada operación dentro de un objeto) y su semántica en lenguaje ingles (que hace cada operación y las reglas de comportamiento). Un proveedor no tiene que suministrar obligatoriamente ningún servicio adicional al ORB, pero si este lo ofrece, deberá esta de acuerdo a la especificación que para este servicio tiene la OMG.

Siempre existen otras maneras de realizar las tareas especificadas por los servicios CORBA™ pero es siempre conveniente seguir las especificaciones de los servicios CORBA™ ya que además de estar basadas siempre en maneras estándar de realizar la tarea especificada, que han demostrado ser eficaces al ser estándar nos permiten ínter operar con implementaciones de terceros.

4.1.1.2. Introducción al servicio de Eventos de CORBA™.

El servicio de eventos CORBA™ permite desacoplar la comunicación entre objetos. En el servicio de eventos se definen dos roles para los objetos, el rol de proveedor (*Supplier*) que corresponde a los objetos que producen información que es tratada por los objetos que realizan el rol consumidor. La información pasada entre proveedores y consumidores se agrupo en conjuntos tratados de forma atómica que denominaremos eventos. Los eventos viajan de los proveedores a los consumidores a través de comunicaciones CORBA™ estándar.

Existen dos maneras estándar de abordar la manera de entablar la comunicación con eventos entre proveedores y consumidores, además también hay dos maneras ortogonales de enviar la información.

Las dos maneras estándar de entablar comunicación con eventos se denominan modelo de inyección y modelo de extracción. El modelo de inyección permite al proveedor comenzar a enviar la información a los consumidores, en el modelo de extracción son los consumidores los que reclaman a los proveedores que les pasen información. En el modelo de inyección es el proveedor el que toma la iniciativa mientras que en el modelo de extracción es el consumidor.

La información en si puede ser genérica o con tipo. En el caso de la información genérica o no con tipo toda la comunicación es a través de operaciones de inyección o extracción que toman un único parámetro que encapsula toda la información del evento. En el caso de ser la información

con tipo las operaciones se definen en OMG™ IDL. Los eventos son pasados aquí a través de parámetros que definimos de cualquier manera. Los canales de eventos con información con tipo no serán tratados en este documento.

Un canal de eventos es un objeto que permite a varios proveedores comunicarse con varios consumidores en modo asíncrono. Un canal de eventos es a la vez un consumidor y un proveedor. Los canales de eventos son objetos estándar CORBA™ y las comunicación con estos se realiza mediante mecanismos estándar de CORBA™.

Principios de diseño:

- Los eventos se emplean en entornos distribuidos. El diseño no dependen de un servicio único, crítico o centralizado.
- El servicio de eventos permite que un evento sea procesado por varios consumidores y tener varios proveedores.
- Los consumidores pueden consumir eventos o ser simplemente notificados de los mismos pudiendo escoger la manera que se desee para el diseño y el desempeño de una aplicación.
- Los consumidores y los proveedores usarán solamente interfaces OMG™ IDL, no se requerirá añadir características extras a CORBA™ para definir esos interfaces.
- Un proveedor tendrá que llamar una sola vez a método estándar para comunicarse con todos los consumidores a la vez.
- Los proveedores podrán emitir eventos sin saber quienes son los consumidores, de igual manera que los consumidores podrán procesar eventos sin saber quienes son los proveedores.
- Los interfaces del servicio de eventos estarán preparados para admitir extensiones referentes a la calidad del servicio. Aunque el servicio de eventos no trata el tema de la calidad del servicio las implementaciones podrán extender los interfaces en función de sus necesidades referentes a este tema.
- Los interfaces del servicio de eventos estarán preparados para ser usados en distintos entornos, como por ejemplo entornos *multihilo* y no *multihilo*.

Consideraciones técnicas:

- Entornos distribuidos: los interfaces están diseñados para permitir tanto a consumidores como a productores desconectarse en cualquier momento, no es necesario un servicio central de identificación, proceso, *enrutado* u otro servicio que pudiera causar un cuello de botella o una caída total del sistema.
- Los eventos no son objetos ya que el modelo de objetos distribuido de CORBA™ no soporta el paso de objetos por valor.
- Generación de Eventos: la especificación de eventos describe como se generan eventos y son transportados de manera general, con un canal de eventos intermedio encargado de transmitirlos entre productores y consumidores. No será necesario realizar sondeos ni que un productor de eventos tenga que notificar a cada receptor.
- "Eventos compuestos": los "eventos compuestos" formados por varios eventos simples pueden ser tratados construyendo un árbol de evaluación de eventos consumidos/enviados valorando sucesivamente las cláusulas cada vez más específicas asociadas

dichos eventos simples. La especificación no exige un servicio de evaluación de predicados global que pudiera perjudicar el rendimiento, eficiencia o seguridad en entornos distribuidos y heterogéneos.

- Examinar, catalogar y filtrar eventos: la especificación usa los mecanismos de CORBA™ de inspección y catalogación para obtener el tipo de los eventos. El filtrado de eventos es sencillo a través de canales de eventos que transporten eventos selectivamente de proveedores a consumidores. Varios canales de eventos pueden trabajar juntos, es decir, un canal de eventos puede consumir eventos suministrados por otro. Los canales de eventos tipados pueden filtrar basándose en el tipo del evento.
- Registro y generación de eventos: los consumidores y proveedores se registran a si mismos en el canal de eventos. Los canales de eventos son objetos que se pueden buscar igual que cualquier otro objeto CORBA™. Un servicio global de registro no es necesario, cualquier evento que implemente el interfaz IDL puede consumir eventos.
- Parámetros de los eventos: la especificación define un parámetro de tipo *any* para enviar un evento, usados para datos específicos de la aplicación.
- Falsificación y eventos seguros: debido a que los productores de eventos son objetos la especificación deja que sean los ORBs los encargados de la seguridad de las referencias de los objetos y las comunicaciones.
- Rendimiento: el diseño es minimalista, se necesita solamente una llamada a través del ORB por evento recibido. Soporta los estilos de inyección y extracción para evitar sondeos por eventos. Como los proveedores, consumidores y canales de eventos son todos objetos del ORB, el servicio se beneficia de cualquier optimización del ORB.
- Formalización de la información de los eventos: para entornos de aplicación específicos y entornos de trabajo puede ser bueno concretizar los datos asociados con un evento (definidos en la especificación con un *any*). Esto puede hacerse especificando una estructura con tipo para esta información. Dependiendo de las necesidades del entorno, puede ser que el tipo de información incluida sea una prioridad, un tiempo, un identificador del origen o un indicador de confirmación. Esta información puede ser únicamente para el consumidor o puede ser tratada por algún canal de eventos.
- Confirmación de la recepción: algunas aplicaciones pueden necesitar que los consumidores confirmen la recepción de los eventos de manera explícita al proveedor. Esto puede realizarse con otro canal de eventos "inverso" en el que los consumidores puedan enviar confirmación como eventos normales. Esto evita la necesidad de un mecanismo especial de confirmación. Puede ser que para un envío atómico estricto entre proveedores y consumidores se requieran interfaces adicionales.

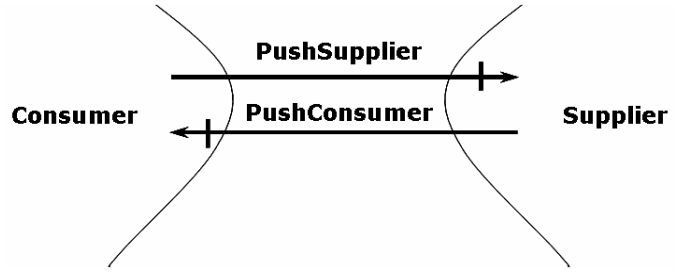
4.1.1.3. Generalidades de la comunicación mediante eventos.

Existen dos modelos para la comunicación entre proveedores y consumidores, el modelo de inyección y el modelo de extracción:

- Modelo de inyección:

En el modelo de inyección los proveedores "inyectan" eventos a los consumidores, es decir, los proveedores suministran eventos a los consumidores invocando operaciones de inyección (*push*) en el interfaz *PushConsumer*.

Para mantener una comunicación mediante el estilo de inyección los consumidores intercambian referencias de objetos *PushConsumer* y *PushSupplier*. La comunicación puede ser interrumpida invocando el método *disconnect_push_consumer* en el interfaz *PushConsumer* o invocando a *disconnect_push_supplier* en el interfaz *PushSupplier*. Si la referencia al interfaz *PushSupplier* es nula (*nil*), la conexión no podrá ser cortada a través del proveedor.

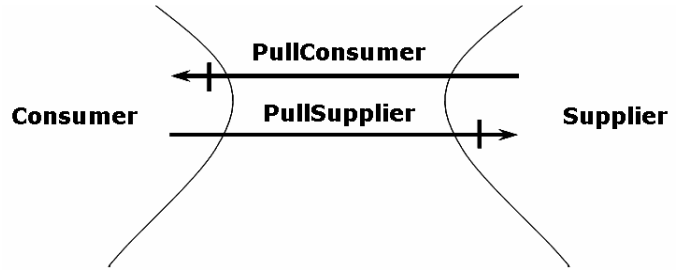


Modelo de inyección.

- Modelo de extracción:

En el modelo de extracción los consumidores "extraen" eventos de los proveedores, es decir, los consumidores solicitan eventos invocando a las operaciones de extracción (*pull*) en el interfaz *PullSupplier*.

Para mantener una comunicación del tipo extracción los consumidores y los proveedores deben de intercambiar las referencias a objetos *PullConsumer* y *PullSupplier*. La comunicación se puede interrumpir llamando al método *disconnect_pull_consumer* del interfaz *PullConsumer* o invocando al método *disconnect_pull_supplier* en el interfaz PullSupplier. Si la referencia del objeto *PullConsumer* es nula (*nil*). La conexión no podrá ser interrumpida en el consumidor.



Modelo de extracción.

4.1.1.4. El módulo CosEventComm.

Los estilos de comunicación que hemos visto antes están soportados por cuatro sencillos interfaces: *PushConsumer*, *PushSupplier*, *PullSupplier* y *PullConsumer*. Estos interfaces se definen en un modulo OMG™ IDL que se llama *CosEventComm*:

```
module CosEventComm {
      exception Disconnected{};
      interface PushConsumer {
            void push (in any data) raises(Disconnected);
            void disconnect_push_consumer();
      };
      interface PushSupplier {
            void disconnect_push_supplier();
      };
      interface PullSupplier {
            any pull () raises(Disconnected);
            any try_pull (out boolean has_event)
                                    raises(Disconnected);
            void disconnect_pull_supplier();
      };
      interface PullConsumer {
            void disconnect_pull_consumer();
      };
};
```

- El interfaz *PushConsumer*:

Un consumidor del tipo inyección deberá de implementar el interfaz *PushConsumer*:

```
interface PushConsumer {
      void push (in any data) raises(Disconnected);
      void disconnect_push_consumer();
};
```

El productor pasará el evento llamando al método *push* y pasando el contenido del evento como parámetro. En caso de que un objeto invoque el método *push* después de haber llamado a *disconnect_push_consumer* la implementación lanzará una excepción del tipo *disconnected*.

El método *disconnect_push_consumer* finalizará la comunicación a través de eventos. Limpiará los recursos usados en la comunicación. La referencia al objeto *PushConsumer* permanecerá disponible. La implementación del método *disconnect_push_consumer* deberá llamar al método *disconnect_push_supplier* del correspondiente interfaz *PushSupplier* (si este interfaz es conocido).

- El interfaz *PushSupplier*:

Un productor de inyección deberá implementar el siguiente interfaz:

```
interface PushSupplier {
      void disconnect_push_supplier();
};
```

El método *disconnect_push_supplier* finaliza la comunicación a través de eventos y limpia todos los recursos necesarios para la misma. La referencia al objeto *PushSupplier* permanecerá disponible.
Al llamar a *disconnect_push_supplier* la implementación del proveedor llamará al método *disconnect_push_consumer* del interfaz *PushConsumer* correspondiente (si este interfaz es conocido).

- El interfaz *PullSupplier*:

Un proveedor de inyección implementará el interfaz *PullSupplier* para enviar eventos.

```
interface PullSupplier {
      any pull () raises(Disconnected);
      any try_pull (out boolean has_event)
                              raises(Disconnected);
      void disconnect_pull_supplier();
};
```

Un consumidor deberá pedir un evento invocando el método *pull* o el método *try_pull* del proveedor. En caso de que un objeto invoque cualquiera de estos dos métodos después de haber llamado a *disconnect_push_supplier* la implementación lanzará una excepción del tipo *disconnected*.

El método pull es *bloqueante*, esperará hasta que exista algún evento y devolverá el valor del mismo.

El método *try_pull* es no *bloqueante*, si no hay evento dará el valor *false* al parámetro *has_event* y devolverá un evento del tipo *long* con un valor indefinido.

El método *disconnect_pull_supplier* finaliza la comunicación a través de eventos y limpia todos los recursos necesarios para la misma. La referencia al objeto *PullSupplier* permanecerá disponible. Al llamar a *disconnect_pull_supplier* la implementación del proveedor llamará al método *disconnect_pull_consumer* del interfaz *PullConsumer* correspondiente (si este interfaz en conocido).

- El interfaz *PullConsumer*:

Un consumidor de extracción implementará el interfaz *PullConsumer*:

```
interface PullConsumer {
      void disconnect_pull_consumer();
};
```

El método *disconnect_pull_consumer* finalizará la comunicación a través de eventos. Limpiará los recursos usados en la comunicación. La referencia al objeto *PullConsumer* permanecerá disponible. La implementación del método *disconnect_pull_consumer* deberá llamar al método *disconnect_pull_supplier* del correspondiente interfaz *PullSupplier* (si este interfaz es conocido).

❖ Observaciones sobre operaciones de desconexión:

Cuando se llama a una operación de desconexión en un proveedor o en un consumidor hay que tener cuidado de que cuando este llame a la operación de desconexión del consumidor o proveedor correspondiente no se produzca una recursividad infinita. Las implementaciones deberán encargarse de esto. Cuando un consumidor o un productor reciban una llamada de desconexión cuando ya habían recibido otra deberán lanzar la excepción *CORBA::OBJECT_NOT_EXIST*.

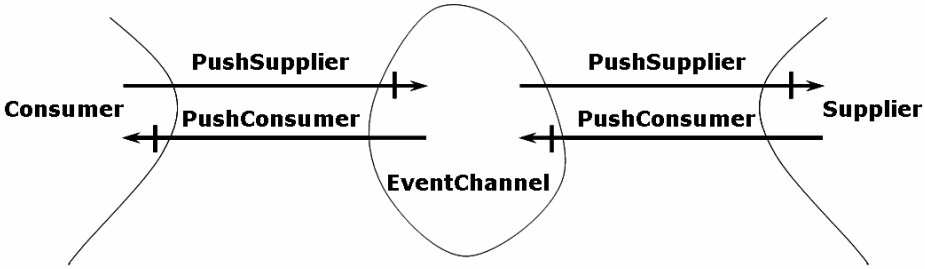
4.1.1.5. Comunicación con canales de eventos.

El canal de eventos es un servicio que desacopla la comunicación entre productores y consumidores. El canal de eventos en si es a la vez un consumidor y un productor de eventos.

Un canal de eventos puede proporcionar comunicaciones asíncronas de eventos entre consumidores y productores. Aunque los consumidores y los productores se comuniquen con el canal usando llamadas a procedimientos estándar de CORBA™, no será necesario que el canal envíe el evento a los consumidores en el instante que lo recibe.

- Comunicaciones del tipo inyección con canal de eventos:

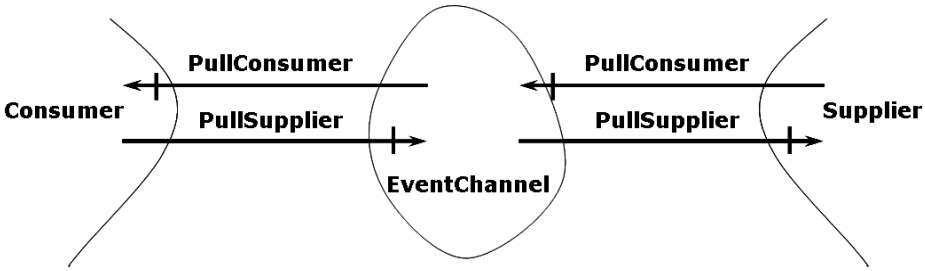
El productor enviará un evento al canal que el canal reenviará al consumidor. El siguiente gráfico ilustra el procedimiento con un productor, un canal de eventos y un consumidor:



Modelo de inyección con canal de eventos.

- Comunicaciones del tipo extracción con un canal de eventos:

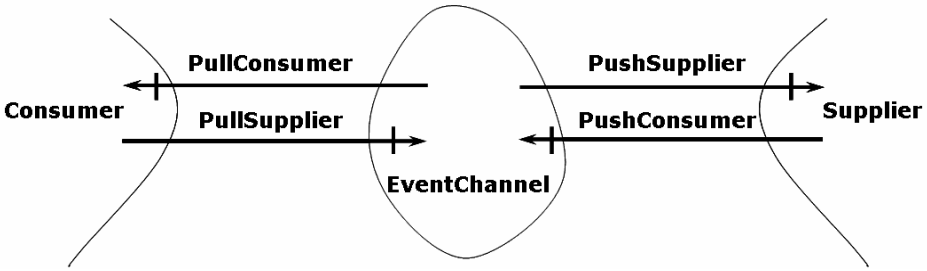
El consumidor extrae eventos del canal, el canal de eventos a su vez extrae eventos del proveedor. La siguiente ilustración muestra este tipo de comunicación con un proveedor, un canal de eventos y un consumidor:



Modelo de extracción con canal de eventos.

- Comunicaciones mixtas con un canal de eventos.

Todos los canales de eventos pueden usar un estilo para comunicarse con un proveedor y un estilo diferente para comunicarse con un consumidor. La siguiente ilustración muestra como un canal de eventos se comunica con el estilo de inyección entre el proveedor y el canal de eventos y usa el estilo de extracción entre el consumidor y el canal de eventos.

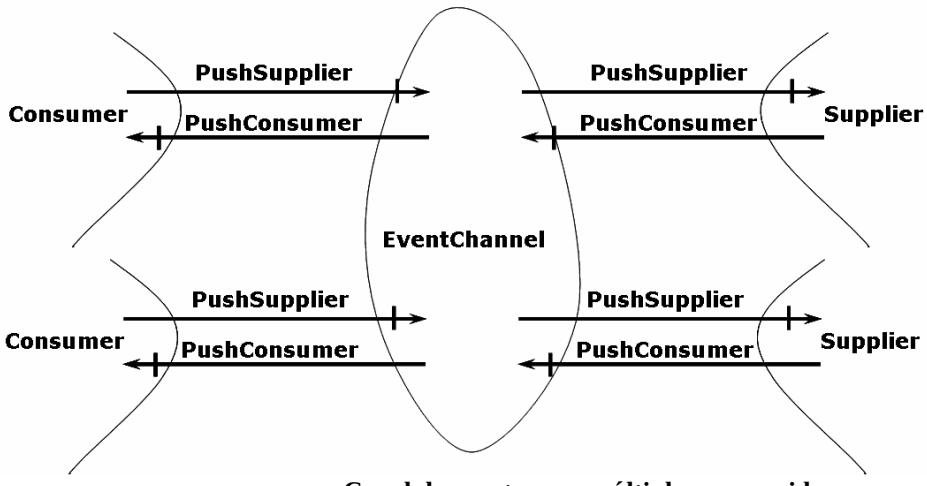


Modelo Mixto.

- Comunicaciones con varios proveedores y varios consumidores.

Las ilustraciones anteriores mostraban a un canal de eventos que se comunicaba con un único proveedor y un único consumidor. Los canales de eventos pueden comunicarse con varios consumidores y varios proveedores. Debido a las especificaciones del servicio un canal de eventos tendrá que enviar un determinado evento a todos los consumidores conectados.

La siguiente ilustración muestra un canal de eventos con varios proveedores de inyección y varios consumidores de inyección:



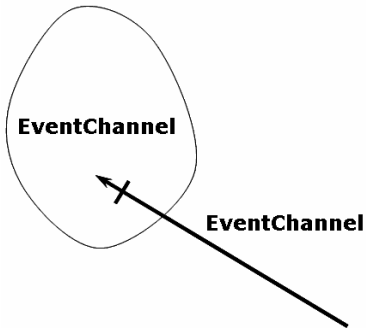
Canal de eventos con múltiples consumidores y proveedores.

Un canal de eventos tiene que ser capaz de soportar consumidores y proveedores que usen distintos tipos de comunicación.

Si un canal de eventos tiene proveedores de extracción continuará extrayendo eventos aunque no tenga consumidores.

4.1.1.6. Administración de un canal de eventos.

Cuando se crea el canal este no tiene ni proveedores ni consumidores. Desde la creación del canal se proporciona a los clientes referencias a un único objeto que implementa el interfaz *EventChannel* tal como muestra la siguiente ilustración:



Canal de eventos sin proveedores ni consumidores.

El canal de eventos define tres métodos de administración: un método que retorna un objeto *ConsumerAdmin* para añadir consumidores, un método que devuelve un objeto *SupplierAdmin* para añadir proveedores y un método para destruir el canal.

Los métodos para añadir consumidores retornan un *proxy* de proveedor. Un *proxy* de proveedor es como un proveedor normal (de hecho el interfaz de *proxys* de proveedor hereda el interfaz de proveedores), con métodos adicionales para conectar al *proxy* de proveedor consumidores.

Los métodos para añadir proveedores retornan un *proxy* de consumidor. Un *proxy* de consumidor es como un consumidor normal (de hecho el interfaz de *proxys* de consumidor hereda el interfaz de consumidores), con métodos adicionales para conectar al *proxy* de consumidor proveedores.

Registrar un consumidor o un proveedor es un proceso con dos pasos. Una aplicación que genera eventos primero obtiene un *proxy* de consumidor de un canal, luego se conecta al *proxy* de consumidor proporcionándole un proveedor. De igual manera una aplicación que consume eventos primero obtendrá un *proxy* de proveedor de un canal y luego se conectará a este *proxy* proporcionándole un consumidor.

El motivo por el que el mecanismo de registro consta de dos pasos es permitir concatenar canales de eventos a través de un agente externo. De esta manera un cliente puede componer dos canales obteniendo un *proxy* de proveedor de uno y un *proxy* de consumidor del otro y pasándole a cada uno la referencia al otro a través de sus respectivos métodos de conexión.

Los *proxys* pueden estar en el estado desconectado, conectado y destruido. Las operaciones de inyección o extracción solamente son válidas en el estado conectado.

4.1.1.6. El modulo CosEventChannelAdmin.

El módulo *CosEventChannelAdmin* contiene interfaces para realizar las conexiones entre proveedores y consumidores. El módulo se define en OMG™ IDL de la siguiente manera:

```
#include “CosEventComm.idl”
module CosEventChannelAdmin {
      exception AlreadyConnected {};
      exception TypeError {};
      interface ProxyPushConsumer: CosEventComm::PushConsumer {
            void connect_push_supplier( in
```

```
                  CosEventComm::PushSupplier push_supplier)
                                    raises(AlreadyConnected);
      };
      interface ProxyPullSupplier: CosEventComm::PullSupplier {
            void connect_pull_consumer( in
                  CosEventComm::PullConsumer pull_consumer)
                                    raises(AlreadyConnected);
      };
      interface ProxyPullConsumer: CosEventComm::PullConsumer {
            void connect_pull_supplier( in
                  CosEventComm::PullSupplier pull_supplier)
                              raises(AlreadyConnected,TypeError);
      };
      interface ProxyPushSupplier: CosEventComm::PushSupplier {
            void connect_push_consumer( in
                  CosEventComm::PushConsumer push_consumer)
                        raises(AlreadyConnected, TypeError);
      };
      interface ConsumerAdmin {
            ProxyPushSupplier obtain_push_supplier();
            ProxyPullSupplier obtain_pull_supplier();
      };
      interface SupplierAdmin {
            ProxyPushConsumer obtain_push_consumer();
            ProxyPullConsumer obtain_pull_consumer();
      };
      interface EventChannel {
            ConsumerAdmin for_consumers();
            SupplierAdmin for_suppliers();
            void destroy();
      };
};
```

- El interfaz *EventChannel*:

El interfaz *EventChannel* define tres métodos administrativos: añadir consumidores, añadir proveedores y destruir el canal:

```
interface EventChannel {
      ConsumerAdmin for_consumers();
      SupplierAdmin for_suppliers();
      void destroy();
};
```

Cualquier objeto que posea una referencia a un objeto que implemente *EventChannel* puede realizar estas operaciones:

- El interfaz *ConsumerAdmin* permite a los consumidores conectarse al canal. El método *for_consumers* devuelve una referencia a un objeto que implementa el interfaz *ConsumerAdmin*.
- El interfaz *SupplierAdmin* permite a los proveedores conectarse al canal. El método *for_suppliers* devuelve una referencia a un objeto que implementa el interfaz *SupplierAdmin*.
- El método *destroy* elimina el canal. La eliminación del canal elimina todos los objetos *ConsumerAdmin* y *SupplierAdmin* creado a través del canal. La eliminación de un objeto *ConsumerAdmin* o *SupplierAdmin* lleva consigo que se invoque el método *disconnect* en todos los *proxys* que hubieran sido creados a través de esos objetos *ConsumerAdmin* o *SupplierAdmin*.

Los objetos para administrar proveedores y consumidores se definen como objetos diferentes de manera que el canal pueda controlar la adición de proveedores y consumidores.

- El interfaz *ConsumerAdmin*:

El interfaz *ConsumerAdmin* define el primer paso para la conexión de consumidores a un canal de eventos; los clientes los usan para obtener *proxys* de proveedores:

```
interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier();
      ProxyPullSupplier obtain_pull_supplier();
};
```

El método *obtain_push_supplier* devuelve un objeto *ProxyPushSupplier*. El objeto *ProxyPushSupplier* se usará posteriormente para conectar un consumidor de inyección.

El método *obtain_pull_supplier* devuelve un objeto *ProxyPullSupplier*. El objeto *ProxyPullSupplier* se usará posteriormente para conectar un consumidor de extracción.

- El interfaz *SupplierAdmin*:

El interfaz *SupplierAdmin* define el primer paso para la conexión de proveedores a un canal de eventos; los clientes los usan para obtener *proxys* de consumidores:

```
interface SupplierAdmin {
      ProxyPushSupplier obtain_push_consumer();
      ProxyPullSupplier obtain_pull_consumer();
};
```

El método *obtain_push_consumer* devuelve un objeto *ProxyPushConsumer*. El objeto *ProxyPushConsumer* se usará posteriormente para conectar un proveedor de inyección.

El método *obtain_pull_consumer* devuelve un objeto *ProxyPullConsumer*. El objeto *ProxyPullConsumer* se usará posteriormente para conectar un proveedor de extracción.

- El interfaz *ProxyPushConsumer*:

El interfaz *ProxyPushConsumer* específica el segundo paso de la conexión de proveedores de inyección a un canal de eventos:

```
interface ProxyPushConsumer: CosEventComm::PushConsumer {
      void connect_push_supplier( in
      CosEventComm::PushSupplier push_supplier)
                                    raises(AlreadyConnected);
};
```

Una referencia de objeto nula (*nil*) puede pasarse como parámetro del método *connect_push_supplier*; si se hace esto no se podrá llamar al método *disconnect_push_supplier* en el proveedor; el proveedor podrá ser desconectado del canal sin necesidad de informarle. Si se pasa un objeto no

nulo a *connect_push_supplier*, la implementación llamará a *disconnect_push_supplier* de ese objeto cuando el *ProxyPushConsumer* sea eliminado. Si el *ProxyPushConsumer* está ya conectado a un *PushSupplier* se lanzará la excepción *AlreadyConnected*.

- El interfaz *ProxyPullSupplier*:

El *ProxyPullSupplier* específica el segundo paso en la conexión de un consumidor de extracción:

```
interface ProxyPullSupplier: CosEventComm::PullSupplier {
      void connect_pull_consumer( in
      CosEventComm::PullConsumer pull_consumer)
                                    raises(AlreadyConnected);
};
```

Se puede pasar una referencia de objeto nula (*nil*) como parámetro del método *connect_pull_consumer*; si se hace eso no se podrá llamar al método *disconnect_pull_consumer* en el consumidor; El consumidor podrá desconectarse del canal sin avisar. Si se pasa una referencia de objeto no nula a *connect_pull_consumer*, la implementación llamará al método *disconnect_pulll_consumer* a través de esa referencia cuando se elimine el *ProxyPullSupplier*.

Si el *ProxyPullSupplier* está ya conectado a un *PullConsumer* se lanzará la excepción *AlreadyConnected*.

- El interfaz *ProxyPullConsumer*:

El interfaz *ProxyPullconsumer* específica el segundo paso para conectar a un proveedor de extracción a un canal de eventos:

```
interface ProxyPullConsumer: CosEventComm::PullConsumer {
      void connect_pull_supplier( in
      CosEventComm::PullSupplier pull_supplier)
                              raises(AlreadyConnected,TypeError);
};
```

La implementación deberá llamar al método *disconnect_pull_supplier* a través de la referencia del objeto pasada por parámetro cuando *ProxyPullConsumer* sea eliminado.

Las implementaciones deberán lanzar la excepción *BAD_PARAM*, estándar de CORBA™, si se le pasa una referencia nula (*nil*) como parámero al método *connect_pull_supplier*.

Si el *ProxyPullConsumer* ya está conectado a un *PullSupplier* la excepción *AlreadyConnected* será lanzada.

- El interfaz *ProxyPushSupplier*:

El interfaz *ProxyPushSupplier* específica el segundo paso para la conexión de un consumidor de inyección a un canal de eventos:

```
interface ProxyPushSupplier: CosEventComm::PushSupplier {
      void connect_push_consumer( in
                  CosEventComm::PushConsumer push_consumer)
                        raises(AlreadyConnected, TypeError);
};
```

La implementación llamará a *disconnect_push_consumer* a través de la referencia de objeto pasada por parámetro a *connect_push_consumer* cuando *ProxyPushSupplier* sea eliminado.

Las implementaciones deberán lanzar la excepción *BAD_PARAM*, estándar de CORBA™, si una referencia nula (*nil*) se pasa como parámetro al método *connect_push_consumer*.

Si el ProxyPushSupplier está ya conectado a un *PushConsumer* se lanzará la excepción *AlreadyConnected*.

4.1.2. Diagrama de Clases de la implementación.

Hemos dividido el diagrama de clases de la implementación del canal de eventos en varias ilustraciones. Esta decisión se ha tomado simplemente para poder imprimir el gráfico en hojas con el mismo formato del resto del documento.

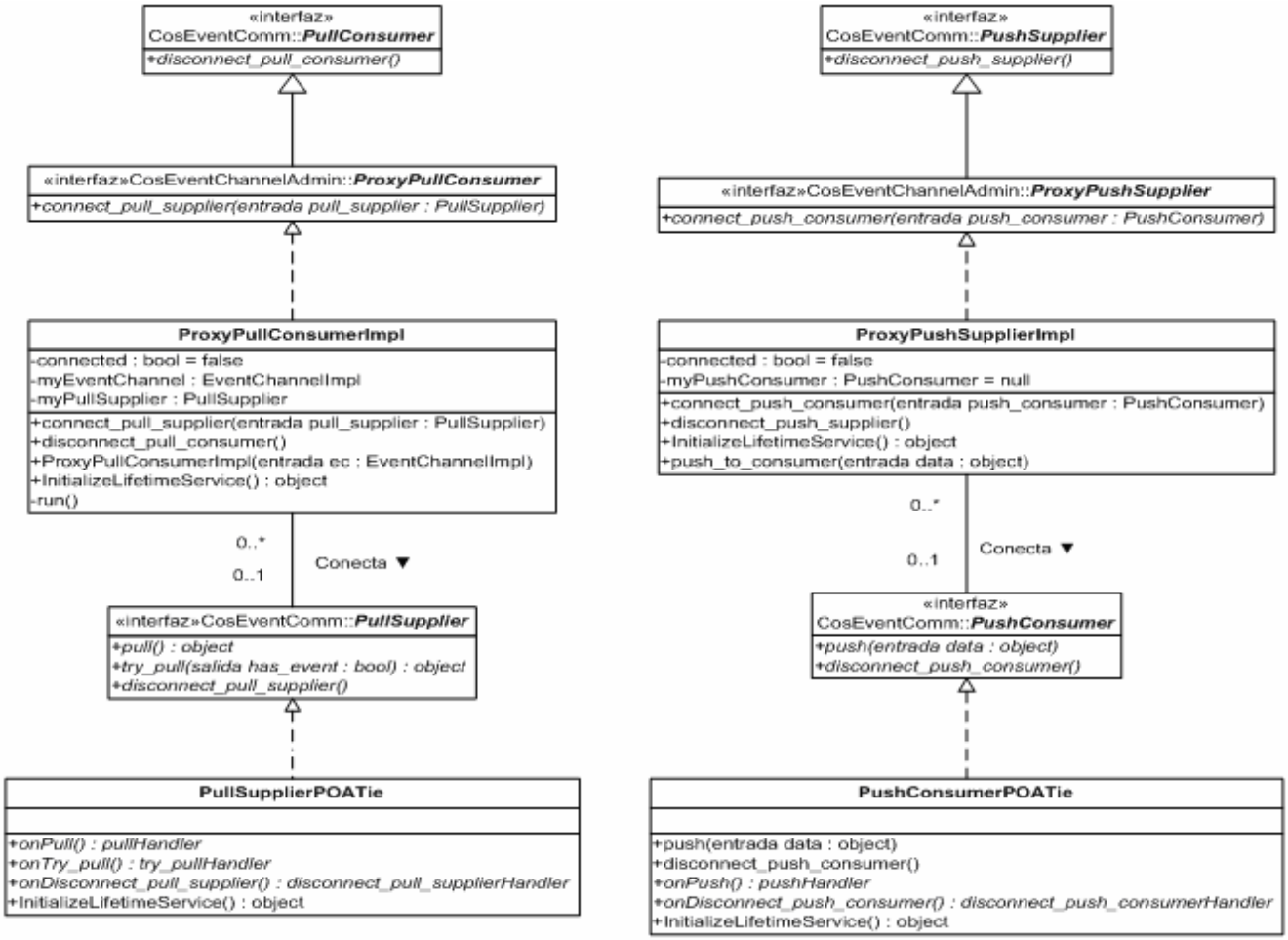


Diagrama de clases del canal de eventos – 1.

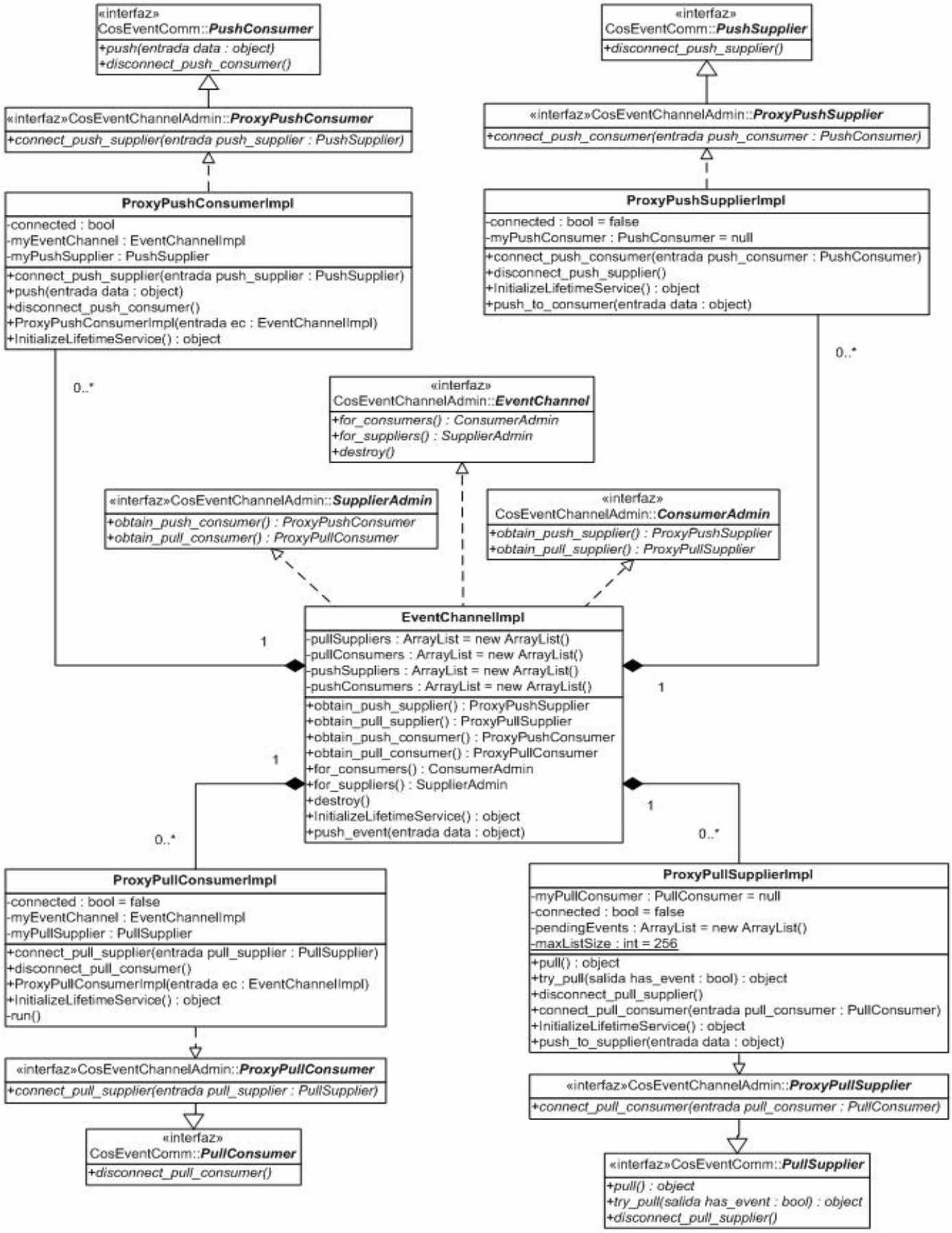


Diagrama de clases del canal de eventos – 2.

4.1.3. Especificación de las clases de la implementación.

La especificación completa a nivel de método puede encontrarse en el CD que acompaña al proyecto.

4.1.3.1. Espacio de nombres omg.org.CosEventComm.

Clases

Clase	Descripción
Disconnected	Excepción que se lanza cuando se trata de hacer una operación de inyección o extracción de eventos cuando la comunicación ha terminado.

Interfaces

Interfaz	Descripción
PullConsumer	Interfaz de consumidores de extracción.
PullSupplier	Interfaz de proveedores de extracción.
PushConsumer	Interfaz de consumidores de inyección.
PushSupplier	Interfaz de proveedores de inyección.

4.1.3.2. Espacio de nombres omg.org.CosEventChannelAdmin.

Clases

Clase	Descripción
AlreadyConnected	Excepción lanzada cuando se trata de conectar un proveedor o consumidor a un *proxy* que previamente ya se había conectado.
TypeError	Excepción usada por los canales de eventos tipados cuando se trata de usar implementaciones de *ProxyPullConsumer* y *ProxyPushSupplier* cuando se les tratan de conectar interfaces de proveedores de extracción y de consumidores de inyección que no incorporan unos determinados requisitos. No se usa en esta implementación.

Interfaces

Interfaz	Descripción
ConsumerAdmin	Interfaz de una factoría de *proxys* de proveedor.
EventChannel	Interfaz de canales de eventos.
ProxyPullConsumer	Interfaz de *proxys* de consumidores de extracción.
ProxyPullSupplier	Interfaz de *proxys* de proveedores de extracción.
ProxyPushConsumer	Interfaz de *proxys* de consumidores de inyección.
ProxyPushSupplier	Interfaz de *proxys* de proveedores de inyección.
SupplierAdmin	Interfaz de una factoría de *proxys* de consumidor.

4.1.3.3. Espacio de nombres es.uc3m.inf.arcos.EventChannel.

Clases

Clase	Descripción
EventChannelImpl	Implementación de un canal de eventos.
ProxyPullConsumerImpl	Implementación de un *proxy* de un consumidor de extracción.
ProxyPullSupplierImpl	Implementación de un *proxy* de un proveedor de extracción.
ProxyPushConsumerImpl	Implementación de un *proxy* de consumidor de inyección.
ProxyPushSupplierImpl	Implementación de un *proxy* de proveedor de inyección.
PullSupplierPOATie	Implementa un proveedor de extracción que lanza Eventos cada vez que se recibe una notificación del canal.
PushConsumerPOATie	Implementa un consumidor de inyección que lanza Eventos cada vez que se recibe una notificación del canal.

Delegados

Delegado	Descripción
disconnect_pull_ supplierHandler	Delegado a través del que el canal avisa de la desconexión del proveedor.
disconnect_push_ consumerHandler	Delegado a través del que el canal avisa de la desconexión del consumidor.
pullHandler	Delegado con el que tratar peticiones *bloqueantes* de eventos desde el canal.
pushHandler	Delegado con el que tratar una recepción de un evento
try_pullHandler	Delegado con el que tratar peticiones no *bloqueantes* de eventos desde el canal.

4.1.4. Diagramas de secuencia de métodos importantes.

- **es.uc3m.inf.arcos.EventChannel.ProxyPullConsumer.Push.**

  Envía un evento al canal de eventos.

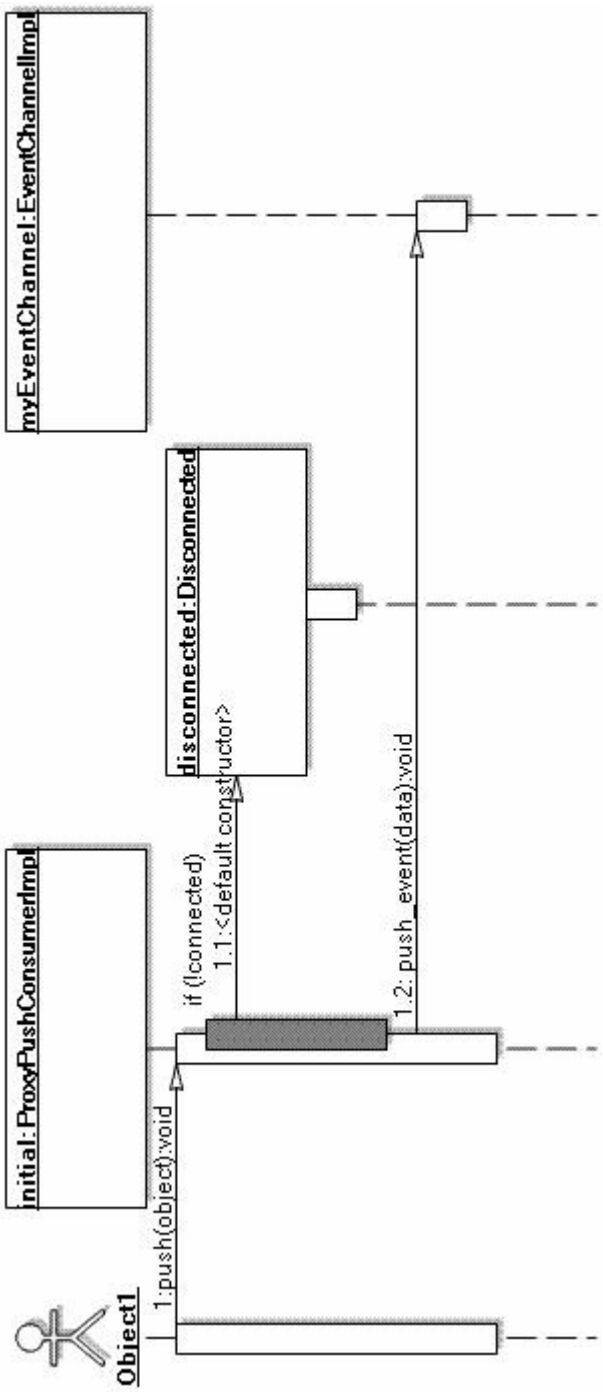


Diagrama de secuencia de Push.

- **es.uc3m.inf.arcos.EventChannel. EventChannelImpl.push_event.**

  Envía un evento recibido en el canal a todos los consumidores.

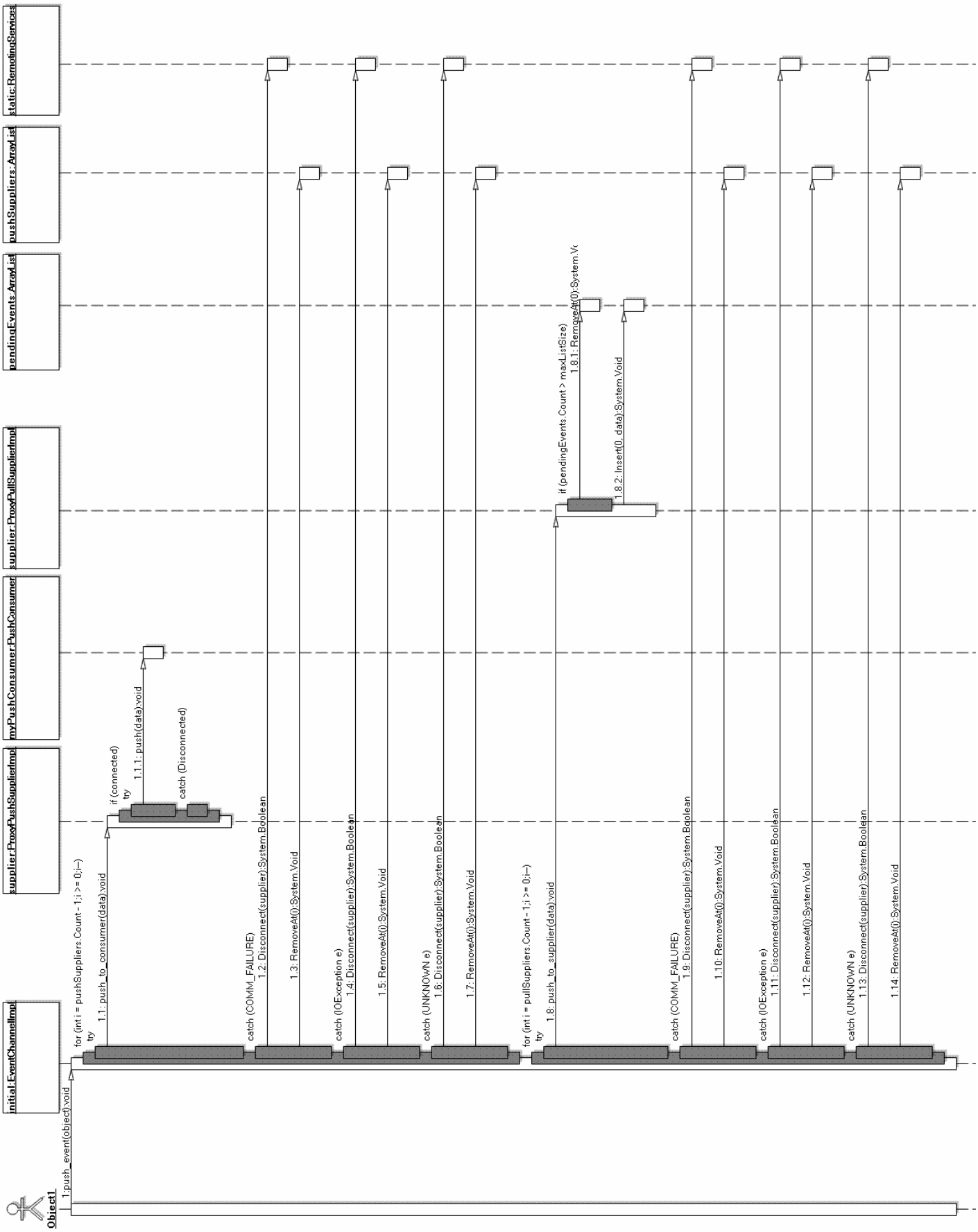


Diagrama de secuencia de Push_event.

- **es.uc3m.inf.arcos.EventChannel.ProxyPullSupplier.Pull.**

Método de la clase *ProxyPullConsumer* para extraer eventos del canal de eventos:

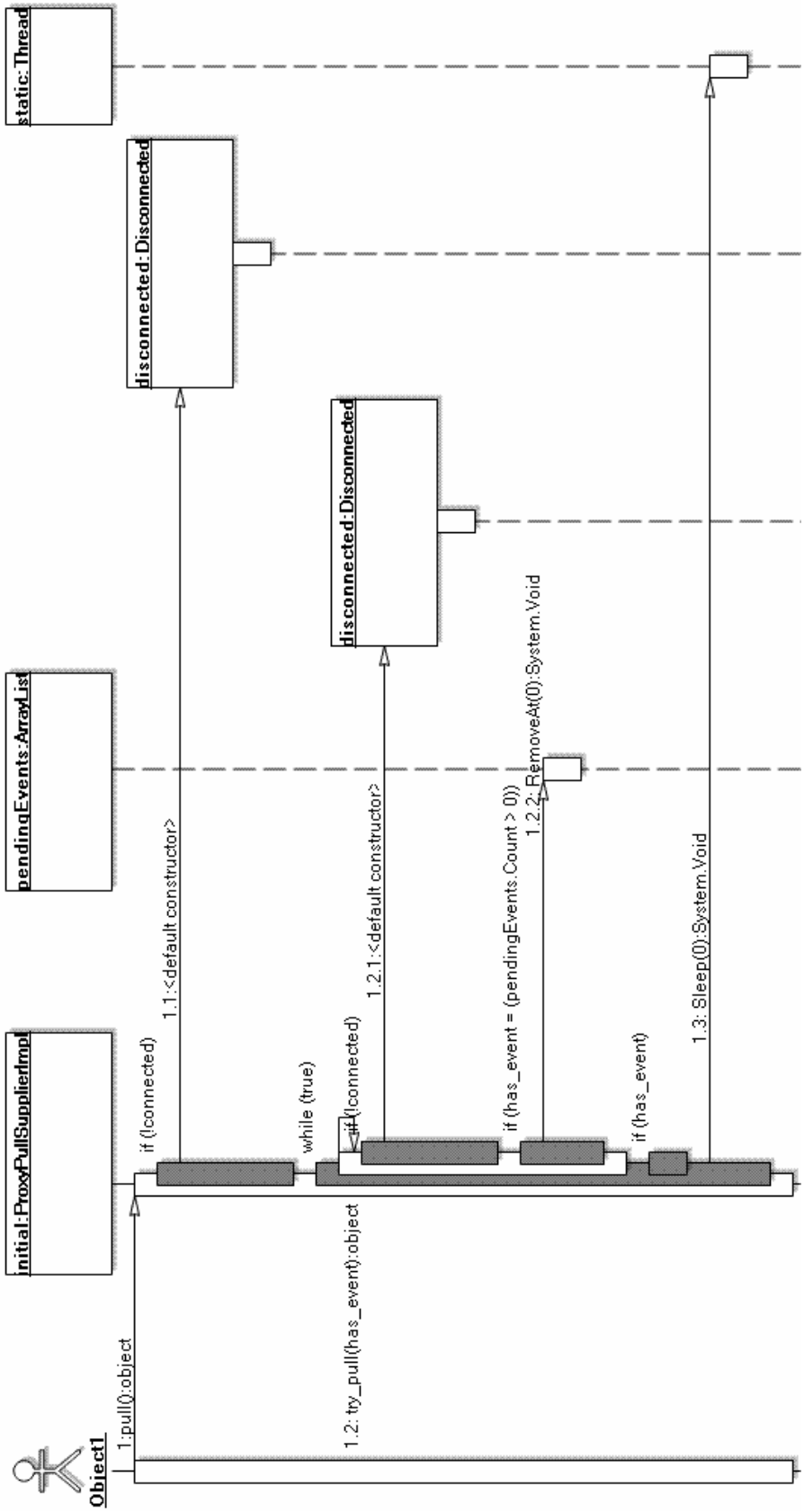


Diagrama de secuencia de Pull.

- **es.uc3m.inf.arcos.EventChannel.ProxyPullSupplier.Run.**

Método principal del hilo que extrae eventos del canal a una lista de eventos pendientes de un *proxy* de consumidor de inyección.

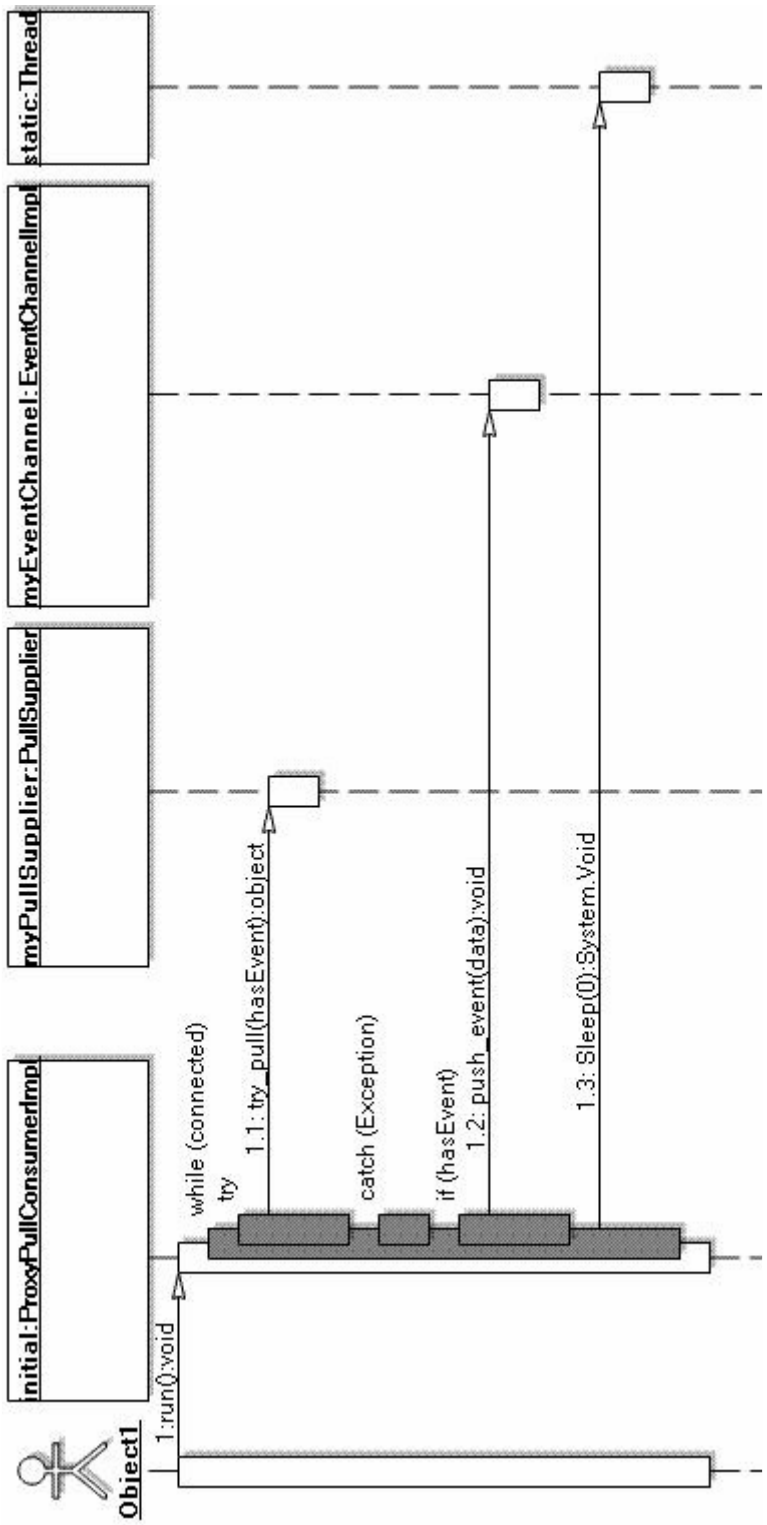


Diagrama de secuencia de Run.

4.2. Desarrollo de un librería cliente/servidor STOMP con C#.

4.2.1. El protocolo STOMP.

4.2.1.1. Introducción a STOMP.

STOMP (Streaming Text Orientated Messaging Protocol) es un protocolo de mensajería diseñado con el fin de poder escribir clientes de manera sencilla en cualquier lenguaje con soporte de sockets. La implementación oficial del servidor STOMP es un conector de ActiveMQ.

Además del servidor STOMP incluido en ActiveMQ existe una implementación completa de STOMP, tanto de servidor como de cliente, en Java™ bajo licencia LGPL llamada SerSTOMP.

Usando como modelo la versión 0.4.0 de SerSTOMP y con la licencia LGPL se ha desarrollado como parte de este proyecto otra implementación completa del protocolo en C# que hemos llamado #STOMP y que describiremos en este apartado.

4.2.1.2. Especificación de STOMP.

Al comienzo de la comunicación el cliente debe abrir un *socket* (se presupone que el *socket* es TCP, pero realmente el tipo del *socket* no es importante). El cliente entonces enviará el siguiente texto:

```
CONNECT
login: <username>
passcode:<passcode>

^@
```

El código ^@ es el carácter nulo (control-@ en ASCII). A partir de ahora llamaremos trama a este tipo de textos. Las tramas comienzan con un comando (en el caso de ser la primera CONNECT), seguido de un salto de línea y una cabecera compuesta por un conjunto de líneas compuestas cada una por pares del tipo <clave>:<valor>. La cabecera terminará con una línea en blanco y a continuación seguirá el cuerpo del mensaje (vacío para el comando CONNECT) terminado en un carácter nulo que indica el final de trama.

Después de que el cliente envíe la trama CONNECT el servidor estará obligado a aceptar la conexión enviando la siguiente trama:

```
CONNECTED
session:<session-id>;

^@
```

El valor de la propiedad sesión será un identificador único (no se usa de momento).

A partir de aquí los tipos de tramas que el cliente puede enviar son SEND, SUBSCRIBE, UNSUBSCRIBE, BEGIN, COMMIT, ABORT, ACK y DISCONNECT.

- Tramas del cliente:

o SEND:

La trama SEND envía un mensaje a un destino a través del sistema de mensajería, requiere que se especifique en la cabecera el destino mediante la clave *destination*. El cuerpo de SEND es el mensaje a enviar:

```
SEND
destination:/queue/a

hello queue a
^@
```

Esto envía un mensaje al destino */queue/a*. Las direcciones no siguen un formato predeterminado según el protocolo, simplemente se trata de cadenas que identifican en el servidor un destino. SEND soporta en la cabecera la clave *transaction* para indicar transacciones.

Es recomendable incluir en la cabecera la clave *content-length* para especificar la longitud en *bytes* del cuerpo de la trama. Si se incluye la clave *content-length* en la cabecera se leerá el número de *bytes* indicado aunque el cuerpo incluya caracteres nulos. La trama deberá terminar siempre con un carácter nulo, en el caso de no estar especificado *content-lenght* se entenderá que el primer carácter nulo es el final de la trama.

o SUBSCRIBE:

Las tramas SUBSCRIBE se usan para registrase como receptores de mensajes de un destino. Como las tramas SEND se requiere que la clave *destination* esté especificada para indicar a que destino nos suscribimos. A partir de que nos suscribimos cualquier mensaje enviado a este destino será enviado por el servidor al cliente. La clave *ack* es opcional, por defecto vale *auto*:

```
SUBSCRIBE
desination: /queue/foo
ack: client

^@
```

Si el valor de la clave *ack* se fija a *client* se enviaran al cliente solamente después que este los acepte con una trama ACK. Los valores válidos para la clave *ack* son *auto*, valor por defecto si no se incluye la clave, y *client*. El cuerpo de SUBSCRIBE se descarta.

o UNSUBSCRIBE:

Las tramas UNSUBSCRIBE sirven para cancelar suscripciones. Es necesario especificar la clave *destination* en la cabecera para indicar el destino del cual queremos dejar de recibir mensajes:

```
UNSUBSCRIBE
destination: /queue/a

^@
```

El cuerpo de UNSUBSCRIBE se descarta.

- BEGIN:

Las tramas BEGIN se usan para comenzar transacciones. Las transacciones sirven para indicar que queremos enviar o aceptar un conjunto de mensajes de forma atómica:

```
BEGIN
transaction: <transaction-identifier>

^@
```

La clave *transaction* es obligatoria, el identificador de transacción se usará en las tramas SEND, COMMIT, ABORT y ACK para indicar pertenencia a una transacción. El cuerpo de BEGIN se descarta.

- COMMIT:

Las tramas COMMIT se usan para terminar transacciones e indicar que se envíen:

```
BEGIN
transaction: <transaction-identifier>

^@
```

La clave *transaction* es obligatoria para indicar la transacción que queremos terminar y enviar. El cuerpo de COMMIT se descarta.

- ACK:

Las tramas ACK se usan para aceptar la recepción de un mensaje correspondiente a una suscripción con aceptación (ACK) *client*. Cuando un cliente se ha suscrito con una trama SUBSCRIBE cuya clave ACK en la cabecera vale *client* se entiende que el servidor no podrá entender que el cliente ha procesado la trama hasta que no reciba una trama de este tipo. ACK requiere indicar en la cabecera la clave *message-id*, que deberá de contener el mismo valor que la clave *message-id* del mensaje que va a ser aceptado. Además la clave *transaction* se puede especificar para indicar que la aceptación corresponde a una determinada transacción:

```
ACK
message-id: <message-identifier>
transaction: <transaction-identifier>

^@
```

El cuerpo de ACK se descarta. La clave *transaction* es opcional.

- ABORT:

Cancela una transacción en curso:

```
ABORT
transaction: <transaction-identifier>

^@
```

El cuerpo de ABORT se descarta. La clave *transaction* es obligatoria para indicar que transacción se descarta:

- DISCONNECT:

La trama DISCONNECT es la manera *educada* de desconectarse del cliente. Se enviará antes de desconectar el *socket*:

```
DISCONNECT

^@
```

El cuerpo de DISCONNECT se descarta.

- Claves de cabecera estándar:

Algunas claves pueden usarse, y tener un significado especial, con la mayoría de paquetes:

- Receipt:

Cualquier trama que no sea del tipo CONNECT puede especificar en su cabecera la clave *receipt* con un valor aleatorio. Este valor hará que el servidor envíe una trama RECEIPT con una cabecera que incluirá la clave *receipt* con el mismo valor de la trama enviada por el cliente:

```
SEND
destination:/queue/a
receipt:message-12345

Hello a!^@
```

- Tramas de servidor:

El servidor además de la trama CONNECTED puede enviar tramas de los tipos MESSAGE, RECEIPT y ERROR:

- MESSAGE:

Las tramas MESSAGE se usan para transportar mensajes de subscripciones al cliente. Las tramas MESSAGE incluyen en su cabecera la clave *destination* indicando a que dirección fue enviado el mensaje. También incluirán la clave *message-id* con un identificador de mensaje único. El cuerpo contendrá el contenido del mensaje:

```
MESSAGE
destination:/queue/a
message-id: <message-identifier>

hello queue a!^@
```

Es recomendable incluir en la cabecera la clave *content-length* para especificar la longitud en *bytes* del cuerpo de la trama. Si se incluye la clave *content-length* en la cabecera se leerá el número de *bytes* indicado aunque el cuerpo incluya caracteres nulos. La trama deberá terminar siempre con un carácter nulo, en el caso de no estar especificado *content-lenght* se entenderá que el primer carácter nulo es el final de la trama:

- RECEIPT:

Los recibos son enviados desde el servidor al cliente cuando este le pide que confirme una determinada trama. Las tramas incluyen en su cabecera la clave *receipt-id* cuando se especifica también en la trama a confirmar, el valor de *receipt-id* será el mismo en la trama de confirmación y en la confirmada:

```
RECEIPT
receipt-id:message-12345

^@
```

El cuerpo de RECEIPT debe ser vacío.

- ERROR:

El servidor puede enviar tramas de ERROR si algo va mal. Las tramas de error deben de contener la clave message cuyo valor contendrá una breve descripción del error. El cuerpo de la trama puede contener más información o ir vacío:

```
ERROR
message: malformed packet received

The message:
-----
MESSAGE
destination:/queue/a

Hello queue a!^@
```

Es recomendable incluir en la cabecera la clave *content-length* para especificar la longitud en *bytes* del cuerpo de la trama. Si se incluye la clave *content-length* en la cabecera se leerá el número de *bytes* indicado aunque el cuerpo incluya caracteres nulos. La trama deberá terminar siempre con un carácter nulo, en el caso de no estar especificado *content-lenght* se entenderá que el primer carácter nulo es el final de la trama.

4.2.2. Diagrama de Clases de la implementación.

Hemos dividido el diagrama de clases de la implementación del canal de eventos en varias ilustraciones. Esta decisión se ha tomado simplemente para poder imprimir el gráfico en hojas con el mismo formato del resto del documento.

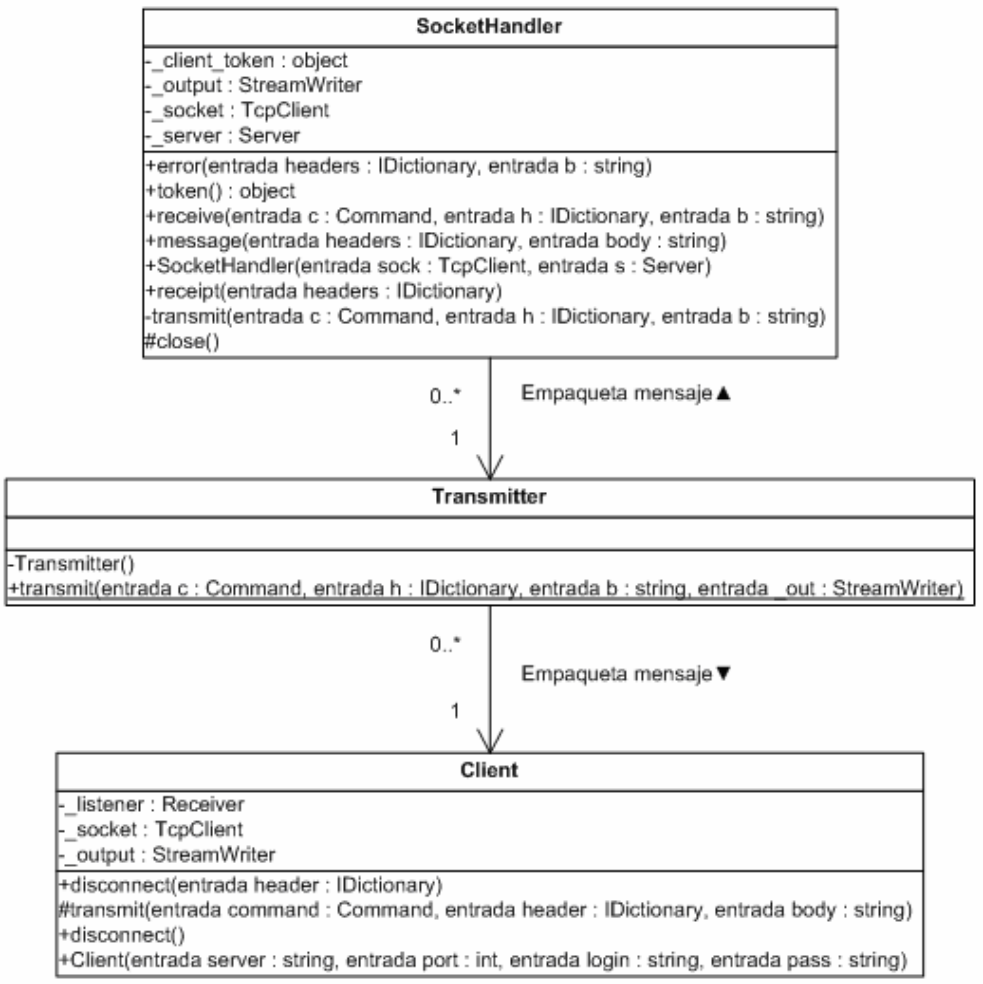


Diagrama de clases de #STOMP -1.

Diagrama de clases de #STOMP -2.

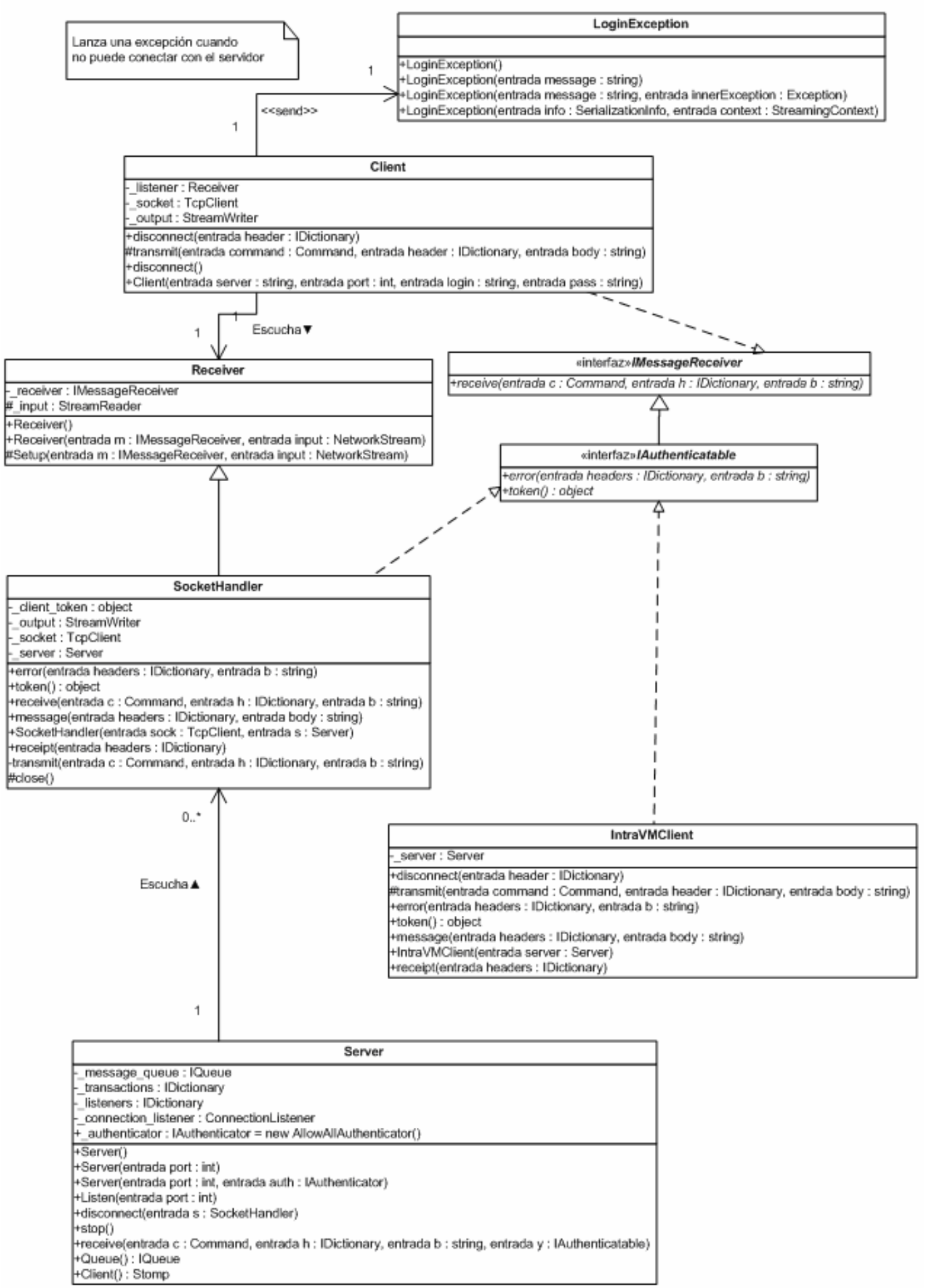


Diagrama de clases de #STOMP -3.

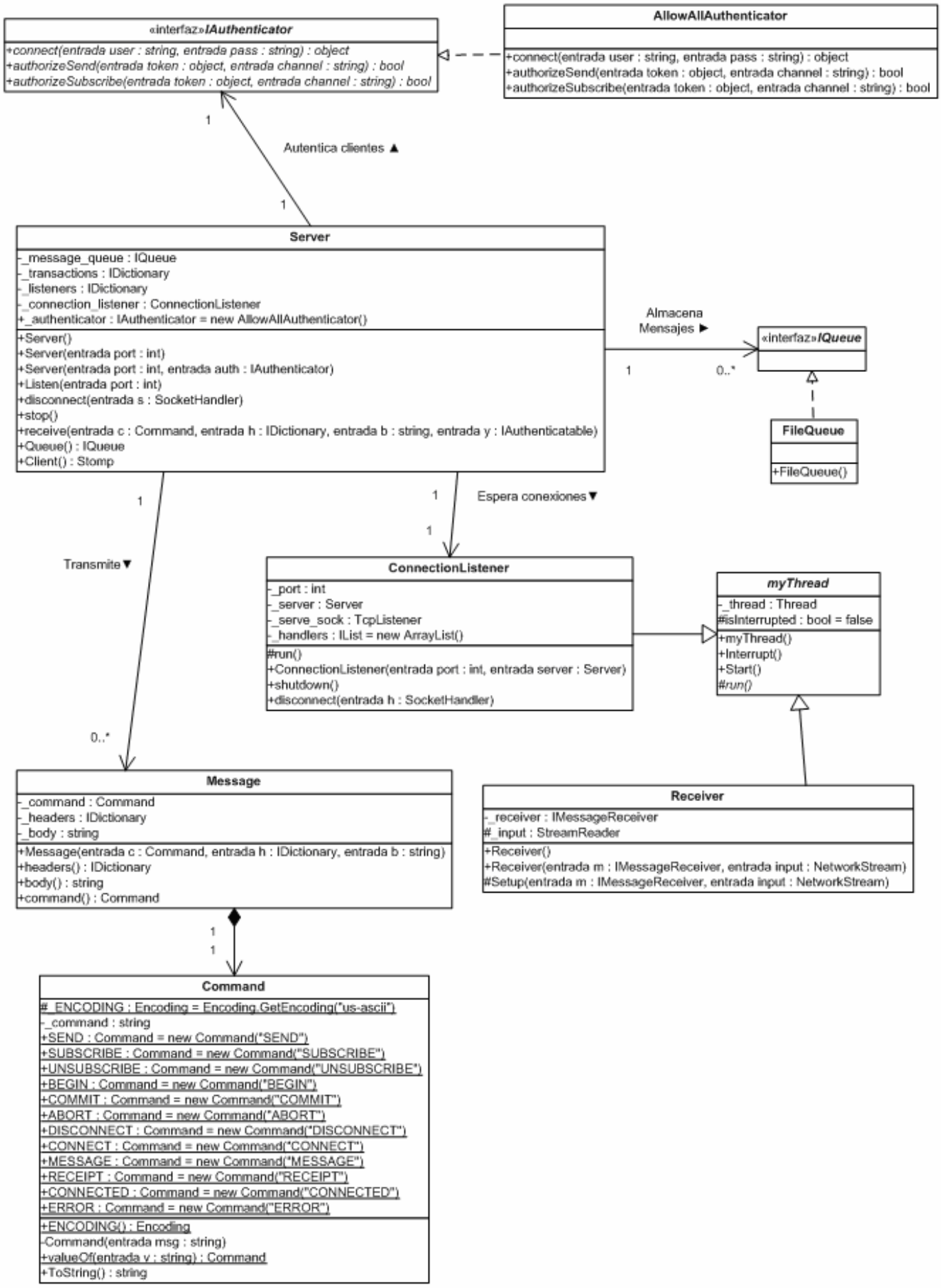


Diagrama de clases de #STOMP -4.

4.2.3. Especificación de las clases de la implementación.

La especificación completa a nivel de método puede encontrarse en el CD que acompaña al proyecto.

4.2.3.1. Espacio de nombres SharpStomp.

Clases

Clase	Descripción
AllowAllAuthenticator	Identificador de clientes que siempre autoriza.
Client	Cliente STOMP .
Command	Comandos de los mensajes.
ConnectionListener	Hilo que espera peticiones de conexión .
FileQueue	Clase que implementa *IQueue*, de momento no hace nada, servirá para en un futuro añadir persistencia a los mensajes a través del sistema de ficheros, implementada por compatibilidad con SerSTOMP.
IntraVMClient	Cliente que se comunica con el servidor a través de la memoria.
ListenerHelper	*Helper* para crear a partir de un delegado para oyentes una instancia de una clase que implemente el interfaz de oyentes.
LoginException	Excepción que se lanza cuando un cliente no se le permite conectarse a un servidor.
Message	Mensaje o trama de STOMP.
myThread	Clase abstracta que imita el comportamiento de la clase *Thread* Java™ permitiendo herencia.
Receiver	Hilo para recibir mensajes y mandárselos a un receptor.
Server	Servidor STOMP.
SocketHandler	Hilo para mantener conexiones con clientes.
Stomp	Implementación del protocolo STOMP Los mensajes se tratan de dos formas. Cuando se llama al método subscribe con un oyente (*listener*) todos los mensajes entrantes se envían a todos los oyentes del canal y son borrados de la cola de mensajes. Si no se proporciona oyente para un canal todos los mensajes son

Clase	Descripción
	almacenados en la cola de mensajes y borrados cuando se soliciten con el método *getNext*.
Transmitter	*Helper* para enviar mensajes.

Interfaces

Interfaz	Descripción
IAuthenticatable	Interfaz para clientes que se han de identificar.
IAuthenticator	Interfaz que han de implementar las clases que se encargan de autorizar usuarios.
IListener	Interfaz para oyentes.
IMessageReceiver	Interfaz que tienen que implementar todas las clases capaces de recibir mensajes.
IQueue	Interfaz de colas para futuros desarrollos, de momento se implementa por compatibilidad con SerSTOMP.

Delegados

Delegado	Descripción
Listener	Delegado para oyentes.

4.3. Desarrollo de una librería cliente de XMLBlaster con XMLRPC.

4.3.1. XMLBlaster.

4.3.1.1. Introducción a XMLBlaster.

XMLBlaster es un *middleware* orientado a mensajes que soporta los modelos publicación/suscripción y punto a punto. Los mensajes se describen a través de meta información codificada en XML.

Los clientes para comunicarse con el servidor pueden usar CORBA™, RMI y XMLRPC. Además XMLBlaster incorpora un interfaz gráfico con el que podemos auditar el servidor.

El API definido mediante OMG™ IDL de XMLBlaster contiene relativamente pocas operaciones, aunque debemos tener en cuenta que muchos parámetros son *string* que contienen documentos XML especificados según los DTD incluidos en la distribución.

4.3.1.2. Interfaces y métodos de XMLBlaster soportados.

- **authenticateIdl.AuthServer**

Interfaz que han de implementar los clientes de XMLBlaster para autenticarse en el servidor.

Interfaces C# que implementan los métodos

Interfaz	Descripción
IClient	Interfaz que han de implementar los clientes de XMLBlaster.

Métodos de instancia públicos

login	Inicia la comunicación con el servidor.
logout	Desconecta el cliente del servidor XMLBlaster y cierra la sesión.

- **ServerIdl.Server**

Interfaz que han de implementar los clientes de XMLBlaster para trabajar con eventos.

Interfaces C# que implementan los métodos

Interfaz	Descripción
IClient	Interfaz que han de implementar los clientes de XMLBlaster.

Métodos de instancia públicos

erase	Borra mensajes del servidor.
get	Obtiene mensajes de manera *síncrona*.
publish	Envía un mensaje al servidor. Dispara el método *update* de los clientes subscritos mediante publicación/subscripción o del designado en una comunicación punto a punto.
publishArr	Envía un *array* de mensajes al servidor. Si algún mensaje no tiene identificador el servidor se lo asigna. Cada mensaje será enviados como un *array* compuesto de la clave del servidor (*string* XML), contenido y calidad de servicio del mismo (*string* XML). Dispara el método *update* de los clientes suscritos mediante publicación/subscripción o del designado en una comunicación punto a punto.
publishOneway	Envía un *array* de mensajes al servidor. Si algún mensaje no tiene identificador el servidor se lo asigna.

	Cada mensaje será enviados como un *array* compuesto de la clave del servidor (*string* XML), contenido y calidad de servicio del mismo (*string* XML). Dispara el método *update* de los clientes suscritos mediante publicación/subscripción o del designado en una comunicación punto a punto. En el caso del protocolo XML-RPC no funciona porque los métodos con XML-RPC están obligados a devolver algo.
subscribe	Suscribe el cliente para que se le notifique de cambios en uno o varios mensajes.
unSubscribe	Cancela una suscripción.

- **clientIdl.BlasterCallback**

Interfaz que han de implementar los clientes para recibir eventos en modo asíncrono del servidor.

Interfaces C# que implementan los métodos

Interfaz	Descripción
ICallback	Interfaz que han de implementar los clientes para recibir eventos en modo asíncrono del servidor.

Métodos de instancia públicos

ping	Comprueba que el servidor de *callback* esté funcionando.
update	Informa de la llegada de un nuevo mensaje cuando se realiza una conexión en modo asíncrono.
updateOneway	Informa de la llegada de un nuevo mensaje cuando se realiza una conexión en modo asíncrono.

4.3.2. Diagrama de Clases de la implementación.

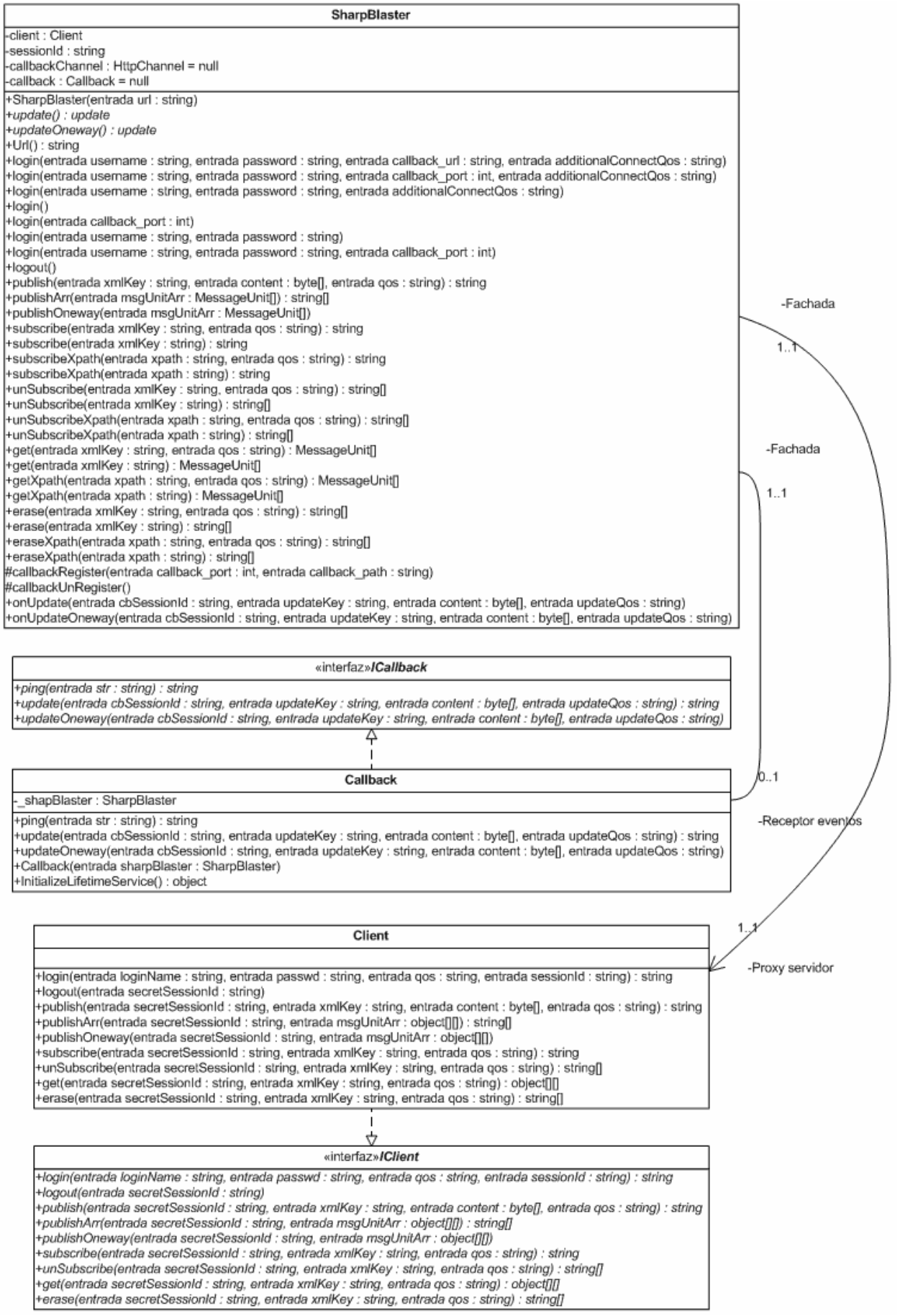


Diagrama de clases de #Blaster.

4.3.3. Especificación de las clases de la implementación.

La especificación completa a nivel de método puede encontrarse en el CD que acompaña al proyecto.

4.3.3.1. Espacio de nombres SharpBlaster.

Clases

Clase	Descripción
Callback	Clase que implementa un servicio XMLRPC para realizar *callback* del servidor XMLBlaster.
Client	Clase que implementa un cliente XMLRPC de XMLBlaster.
MessageUnit	Clase pare leer y escribir *MsgUnit* de XML.
SharpBlaster	Clase a través de la cual un cliente puede acceder y recibir datos de un servidor XMLBlaster con el protocolo XMLRPC

Interfaces

Interfaz	Descripción
ICallback	Interfaz que han de implementar los clientes para recibir eventos en modo asíncrono del servidor.
IClient	Interfaz que han de implementar los clientes de XMLBlaster.

Delegados

Delegado	Descripción
update	Delegado para usar con los eventos del servidor de *callback*.

4.3.4. Diagramas de secuencia de métodos importantes.

- **SharpBlaster.SharpBlaster.login.**

  Inicia la comunicación con el servidor.

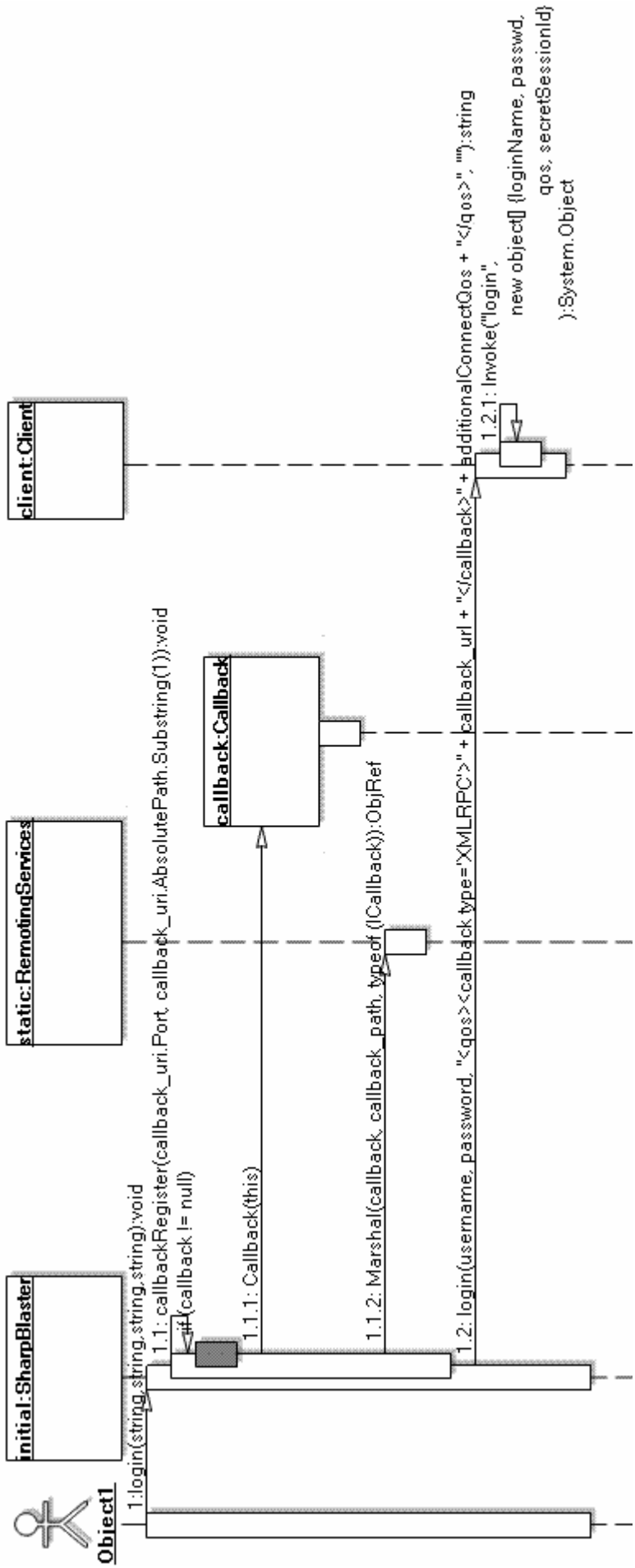


Diagrama de secuencia de Login.

- **SharpBlaster.SharpBlaster.logout.**

  Desconecta el cliente del servidor XMLBlaster y cierra la sesión.

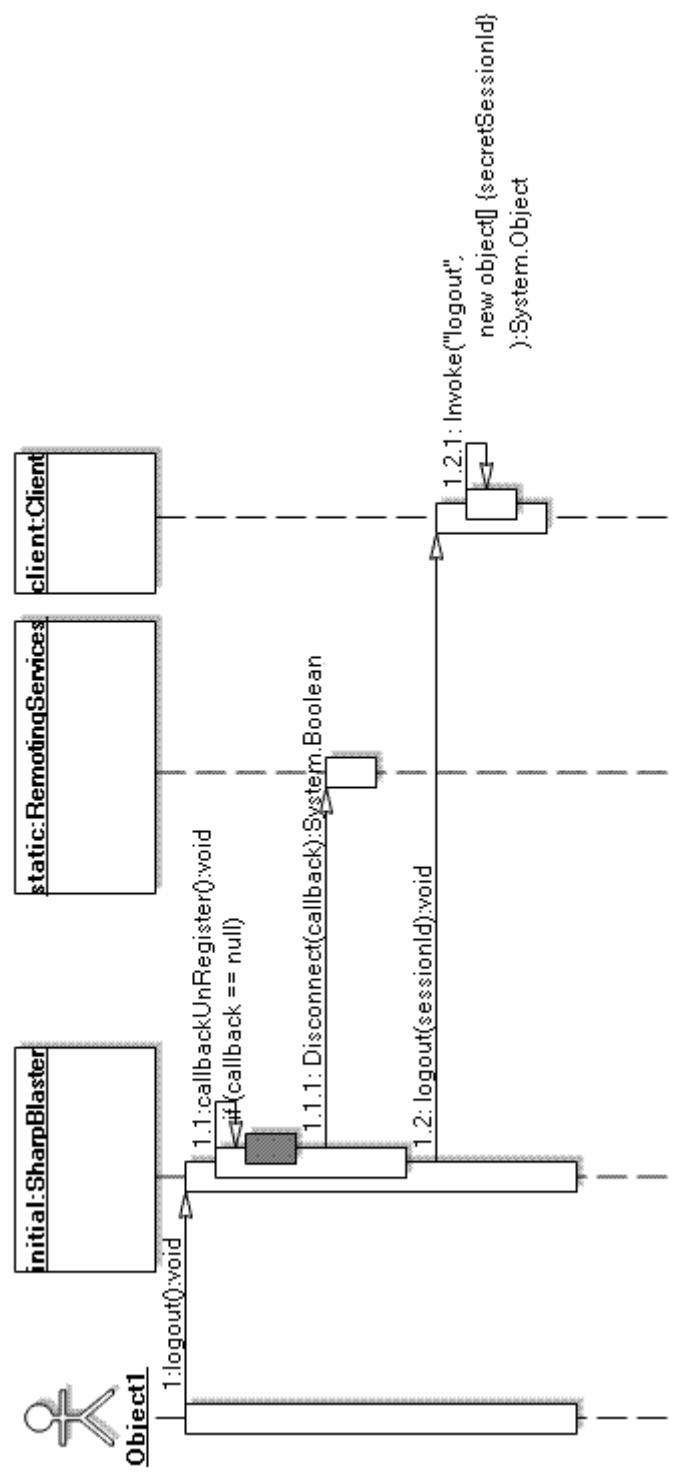


Diagrama de secuencia de Logout.

5. Evaluación de la eficiencia del canal de eventos.

5.1. Objetivos de la evaluación.

En este apartado se va a comparar el rendimiento de las dos implementaciones del servicio EventChannel tratadas en el apartado 3.7.3. de esta documentación que son la de Jacorb y la nuestra propia. El estudio se limita solamente al modelo de inyección dónde no hay implementado ningún sondeo y el paso de mensajes es totalmente *síncrono*.

Para la elaboración de estas pruebas hemos contando con un cluster de 7 máquinas comunicadas a través de tres tipos de red, una Ethernet, una Fast-Ethernet y una Giga-Ethernet.

Al ser las dos implementaciones compatibles no solamente nos vamos a limitar a tratar los casos en que los proveedores, el canal de eventos y los consumidores sean del mismo tipo, sino que también vamos a abordar aquellas en que los proveedores son todos del mismo tipo; los consumidores también son del mismo tipo, aunque pudiendo ser este distinto al de los proveedores y el canal de eventos se ejecute en cualquiera de las dos plataformas.

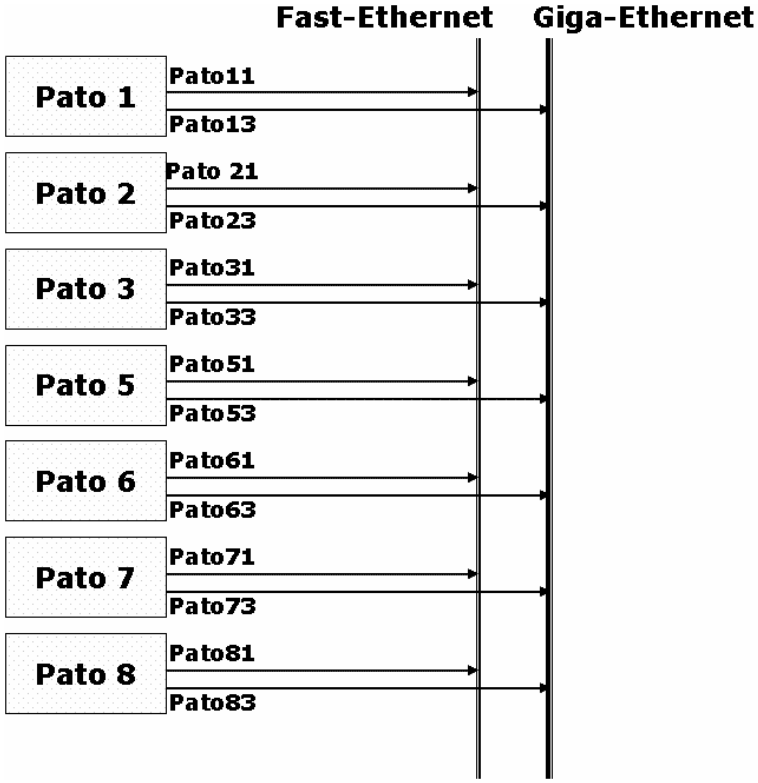
Buscaremos, entre las anteriormente citadas, la mejor combinación posible de proveedores, canal de eventos y consumidores Mono o Java™ con 7 máquinas . Además entre los nodos del cluster hemos usado dos de los tres tipos red: la Fast-Ethernet y la Giga-Ethernet, con los que también estudiaremos el impacto de usar un tipo de red u otro.

5.2. Entorno de pruebas.

5.2.1. Hardware.

Disponemos de un cluster de 7 ordenadores con dos procesadores Pentium III cada uno (*dual-processor*). La frecuencia de cada procesador es de 1Ghz. Cada ordenador dispone de 1 GB de memoria RAM y están interconectados a través tres redes: una red Ethernet, una Fast-Ethernet y una Giga-Ethernet.

Para la realización de esta comparativa solamente se han usado las redes Fast-Etherner y Giga-Ethernet.



Conexión del cluster Pato.

5.2.2. Software.

Sistema Operativo Linux®
- Distribución: GNU/Linux® Debian 3.0
- Kernel: 2.4.20 SMP

Java
- Máquina virtual: Java™ 1.5.0.0_4-b05
- Entorno de desarrollo: Eclipse3.1.1.
- Compilador: Eclipse JDT Batch Compiler 3.1.1. ~ Javac1.5.
- Implementación CORBA™: Sun™ Orb 1.5
- Implementación EventChannel: Jacorb 1.7.1
- Compilador C: Gcc 3.3.5
- Parámetros de compilación en C: -O2 -shared
- Despliegue: FatJar 0.0.20

Mono (CLI)
- Máquina virtual: Mono 1.1.9.1.
- Entorno de desarrollo: Monodevelop 0.7
- Compilador: Mono Mcs 1.1.9.1
- Implementación CORBA™: IIOP.Net 1.7.1
- Implementación EventChannel: EventChannel C#
- Despliegue: Nant 0.85-rc3

5.3. Tiempo de ejecución.

Tanto Java™ como CLI incorporan relojes con los que medir tiempos, aunque motivos de portabilidad la resolución de dichos relojes en ambas plataformas es de milisegundos. Esta resolución es insuficiente para realizar las medidas de esta comparativa.

El sistema operativo Linux® sobre el que realizamos las pruebas ofrece a través de la función Posix *gettimeofday* una resolución en micros Intel de microsegundos. *Gettimeofday* no está presente en el sistema Windows y cualquier solución basada en esta función no seria exportable a Windows®. Como las pruebas van a ser realizadas sobre Linux® en su totalidad la medida de tiempos con *gettimeofday* si es una solución razonable en este caso.

Para acceder a *gettimeofday* debemos de utilizar los mecanismos que ofrecen tanto Mono como Java™ para el acceso a funciones nativas o no manipuladas.

En Mono para el acceso a funciones no manipuladas se usa el mecanismo estándar especificado por CLI y llamado Pinvoke. Con Pinvoke simplemente hemos de conocer las signatura de la función en C y adecuarla a C# indicando mediante anotaciones el lugar dónde se encuentra la implementación de dicha función. Como a esta función se le pasa una estructura también hemos de escribir esa estructura C en C#:

```
private struct timeval
{
      public int tv_sec;
      public int tv_usec;
}

[DllImport("libc")]
private static extern int gettimeofday(out timeval tv
                                              , IntPtr ignore);
```

En Java™ no se pueden llamar a las funciones nativas directamente, es necesario escribir un envoltorio en C con la librería JNI y compilarlo como librería dinámica, ELF en el caso de Linux®. Java™ a través de JNI podrá llamar a este envoltorio en tiempo de ejecución. La implementación del envoltorio es:

```
JNIEXPORT void JNICALL Java_Timeval_callGetTimeOfDay
                              (JNIEnv * env, jobject obj_this)
{
  struct timeval tp;
  if (gettimeofday(&tp, NULL) == -1)
    (*env)->FatalError(env, "gettimeofday call failed.");

   jclass class_Timeval = (*env)->GetObjectClass(env,
                                              obj_this);
   jfieldID id_Tv_sec = (*env)->GetFieldID(env,
                              class_Timeval, "Tv_sec", "I");
   (*env)->SetIntField(env, obj_this, id_Tv_sec, tp.tv_sec);

   jfieldID id_Tv_usec = (*env)->GetFieldID(env,
   class_Timeval, "Tv_usec", "I");
  (*env)->SetIntField(env, obj_this, id_Tv_usec, tp.tv_usec); }
```

En Java™ para llamar a este envoltorio escribiremos:

```
static{
   System.loadLibrary("Timeval");
}
private native void callGetTimeOfDay();
```

Este caso ilustra muy bien el hecho de que CLI ha sido pensado desde un principio para hacer uso de lo ya implementado, mientras que Java™ en un comienzo aboga más realizar nuevas implementaciones *portables*. Ambos planteamientos tienen sus pros y sus contras, por ejemplo las implementaciones *portables* no pueden sacar partido de las mejoras de rendimiento que ofrecen las plataformas sobre las que se ejecutan las máquinas virtuales, a cambio los desarrolladores solamente se ven obligados a mantener una versión apta para su uso con cualquier implementación de máquina virtual de las existentes para esa plataforma de desarrollo.

Como técnica para tratar de que las llamadas a *gettimeofday* influyan lo menos posible vamos a hacer tres tomas de tiempo. Haremos una primera toma de tiempos (A) antes de lo que queramos medir y dos consecutivas después (B y C). El tiempo que devolvamos será el resultado de esta operación:

```
Tiempo Resultante = (tiempo B-tiempo A)-(tiempo C-tiempo B)
```

5.4. IOR en máquinas multihome.

Una máquina *multihome* es aquella que tiene varías direcciones IP. Las máquinas del cluster que estamos usando tienen varías direcciones IP porque tienen varías tarjetas de red. Según la dirección IP o nombre de dominio usado para acceder de un nodo a otro la comunicación se hará a través de Ethernet, Fast-Ethernet o Giga-Ethernet.

En CORBA™ las direcciones de los objetos hospedados se transmiten mediante IORs (Interoperable Object Reference). Las IOR sirven para indicar al servicio de nombres dónde se publica un objeto y también para indicar a los servidores sobre que objeto del cliente deben de hacer una retro llamada.

Un IOR contiene la dirección IP o el nombre de la máquina sobre la que está un determinado ORB. Cuando un cliente quiere llamar a un método de un determinado objeto a través de una red TCP/IP usa el protocolo IIOP™ y la dirección IP o nombre de dominio contenido en la IOR de dicho objeto.

Controlando las direcciones IP o nombres de dominio contenidas en las IOR podremos decidir el tipo de conexión en nuestro cluster que vamos a usar.

La dirección IP o nombre de dominio de una IOR que ha de poner un ORB por defecto no está definida en el estándar CORBA™ y las implementaciones suelen usar la primera dirección IP que les devuelve el sistema operativo. Al igual que no está definida la manera en que se

selecciona tampoco está definida la manera en que el programador fuerza ese valor según sus necesidades.

En el ORB de Sun™ la dirección IP o nombre de dominio de las IOR se puede fijar con la propiedad *com.sun.CORBA.ORBServerHost*:

```
System.setProperty("com.sun.CORBA.ORBServerHost","pato11");
```

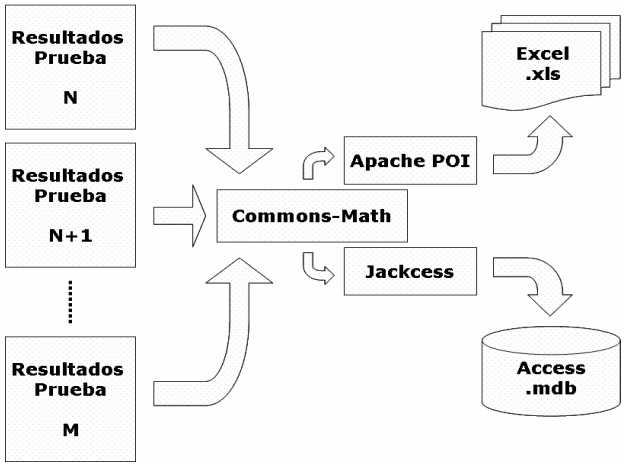
Con IIOP.Net se lo fijamos al crear el canal:

```
IDictionary properties = new Hashtable();
properties[IiopServerChannel.PORT_KEY] = 0; //Random port
properties[IiopServerChannel.MACHINE_NAME_KEY]="pato11";
IiopChannel channel = new IiopChannel(properties);
```

5.5. Recogida y tratamiento de datos.

La gran cantidad de datos recogida hace que no sea viable realizar el análisis de los mismos sobre este tipo de ficheros. En este proyecto hemos optado por realizar un programa Java™ que para cada fichero de texto plano realice la media y la desviación típica de todas sus medidas de tiempo, posteriormente almacenará dichos resultados en una archivo Excel y en una base Access con el resto de medias y desviaciones calculadas a partir de los otros de ficheros.

Para realizar las medias y las desviaciones típicas se ha usado la librería Commons-math, para escribir los archivos Excel se ha utilizado Apache POI y la base de datos Access se ha creado con la librería Jackcess.



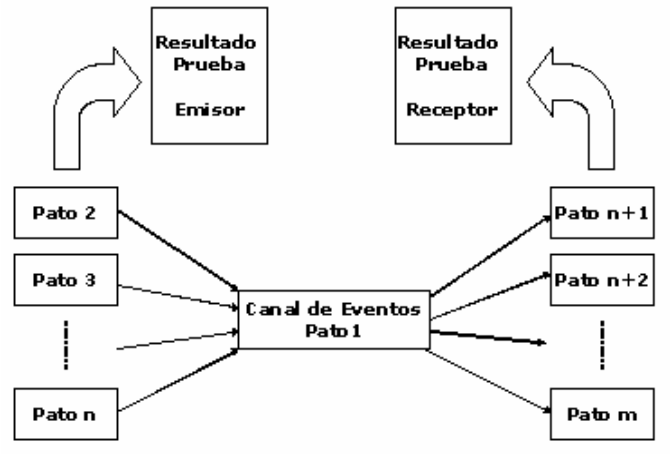
Tratamiento de datos.

5.6. Evaluación del ancho de banda.

Ancho de banda es la cantidad de datos que se pueden transmitir en una unidad de tiempo.

En esta prueba vamos a tratar de medir el tiempo que tarda en enviar una determinada cantidad de eventos un proveedor y también el tiempo que tardan los consumidores en recibir un número de eventos, todo ello en función del número de proveedores y consumidores que hay conectados una canal de eventos.

Cada vez que ejecutemos una prueba de ancho de banda un proveedor y un consumidor escribirán en un fichero el tiempo que tardan en enviar y recibir respectivamente una cierta cantidad de eventos que le indiquemos.



Evaluación del ancho de banda.

El número de eventos seleccionado para la realización de las pruebas es de 1000, suficiente para obtener en cada medición al menos 7 dígitos significativos.

Cada evento contendrá un entero, los enteros ocupan 4 *bytes*.

En el CD que acompaña esta documentación se incluyen todos los resultados obtenidos, en esta memoria nos vamos a quedar con los mejores tiempos.

Lógicamente una solución es mejor que otra cuando el tiempo necesario para enviar o recibir eventos por un proveedor o por un consumidor es más pequeño.

Cada combinación se ha realizado 5 veces y como resultado se ha tomado la media, el coeficiente de dispersión máximo calculado a partir de esas medidas en cada combinación es de un 14%.

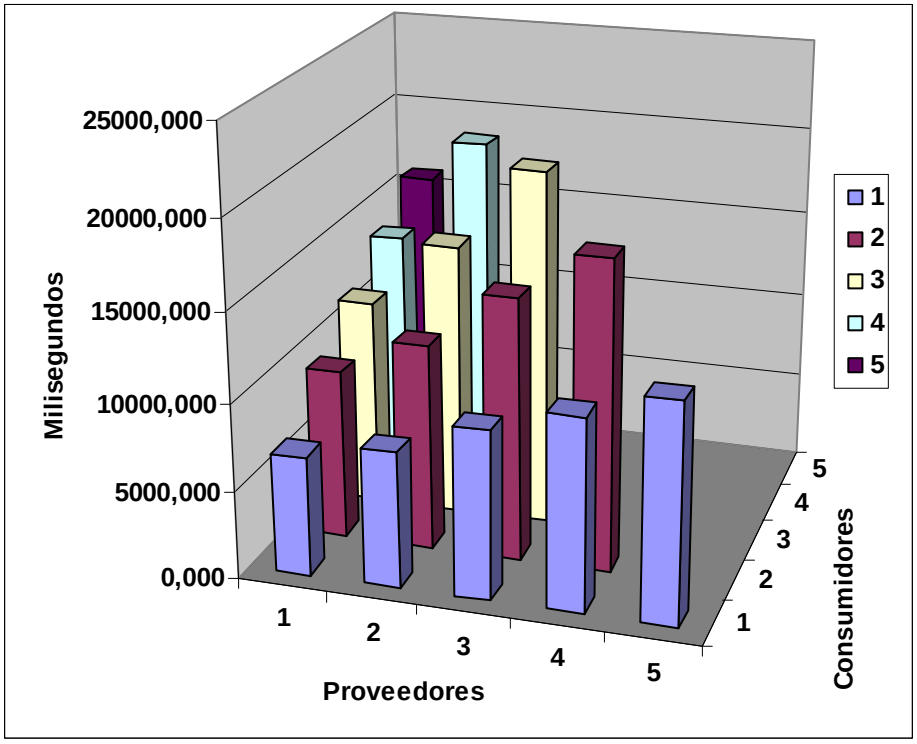
5.6.1. Proveedores con Fast-Ethernet.

Tipo prueba	Ancho de banda
Variable	Proveedor
Número eventos	1000
Interfaz	Fast-Ethernet
Unidad de tiempos	Milisegundos

Color			
Proveedores	Java	Java	Mono
Canal Eventos	Java	Java	Java
Consumidores	Java	Mono	Mono

Mejores		Consumidores				
		1	2	3	4	5
Proveedores	1					
	2					
	3					
	4					
	5					

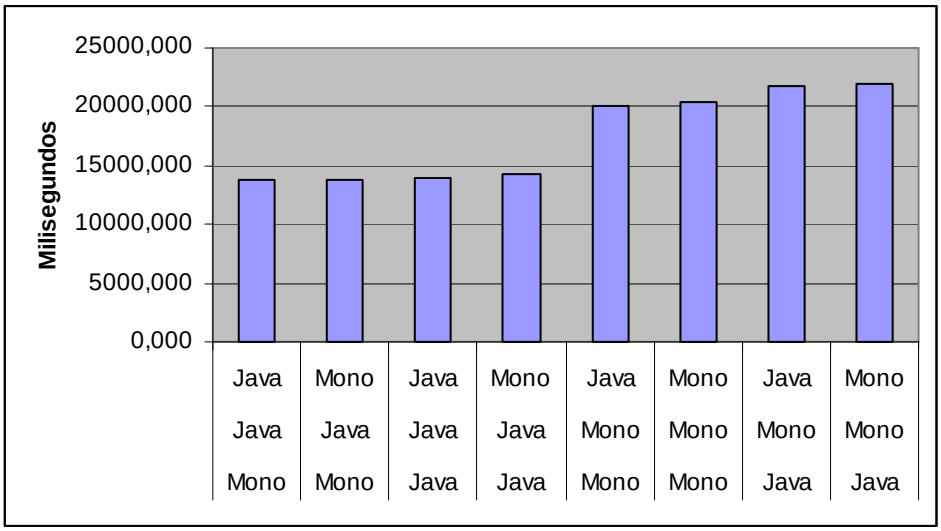
Media		Consumidores				
		1	2	3	4	5
Proveedores	1	6782,218	9615,215	11768,919	13969,286	16018,620
	2	7710,631	11648,682	15548,739	19918,499	
	3	9552,834	14898,508	20242,030		
	4	10915,488	17632,229			
	5	12417,601				



Proveedores con Fast-Ethernet.

Como podemos ver en el gráfico las combinaciones que más se repiten son proveedores Java, canal de eventos Java™ y consumidores Java™ y proveedores Java, canal de eventos Java™ y consumidores Mono con 6 veces cada una. A continuación en el ranking podremos ver que la combinación más rápida es proveedores Java, canal de eventos Java™ y consumidores Mono. La peor combinación es un 60% más lenta que la mejor, el canal de eventos Mono es en media un 51% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	13692,848
2	Mono	Java	Mono	13833,569
3	Java	Java	Java	13916,833
4	Mono	Java	Java	14339,655
5	Java	Mono	Mono	20099,203
6	Mono	Mono	Mono	20377,555
7	Java	Mono	Java	21849,206
8	Mono	Mono	Java	21970,405



Comparativa de proveedores con Fast-Ethernet.

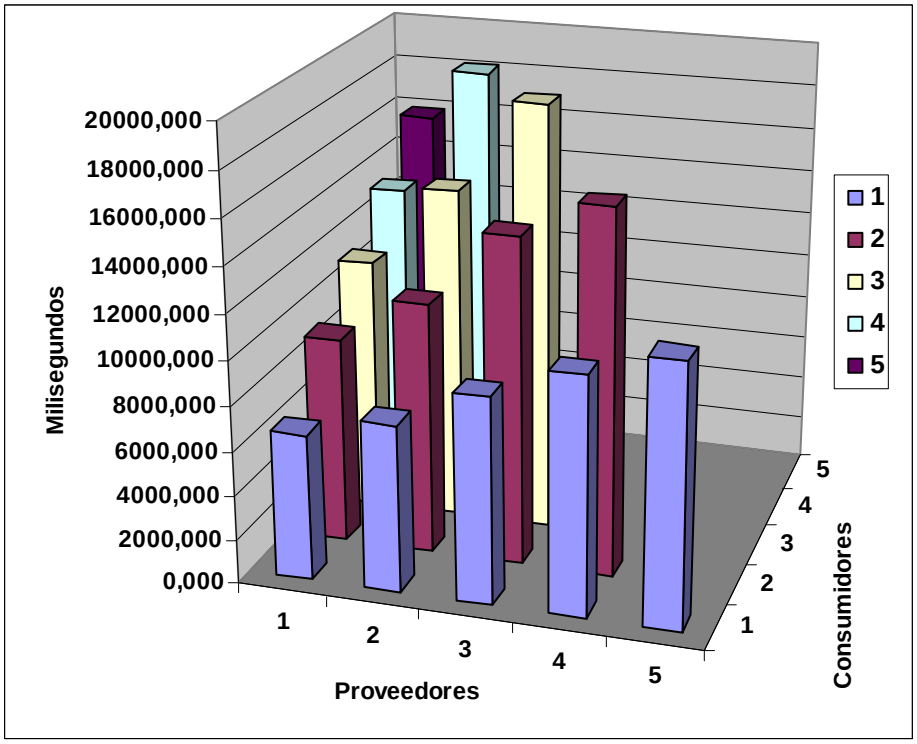
5.6.2. Proveedores con Giga-Ethernet.

Tipo prueba	Ancho de banda
Variable	Proveedor
Número eventos	1000
Interfaz	Giga-Ethernet
Unidad de tiempos	Milisegundos

Color				
Proveedores	Java	Java	Mono	Mono
Canal Eventos	Java	Java	Java	Java
Consumidores	Java	Mono	Java	Mono

Mejores		Consumidores				
		1	2	3	4	5
Proveedores	1					
	2					
	3					
	4					
	5					

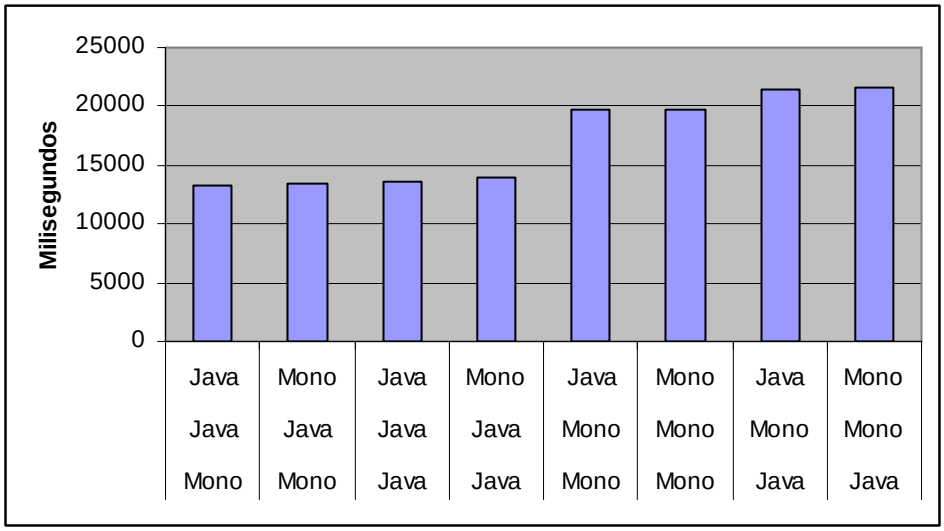
Media		Consumidores				
		1	2	3	4	5
Proveedores	1	6571,447	9323,560	11419,849	13517,731	15857,817
	2	7506,572	11319,155	15054,170	19209,824	
	3	9255,958	14693,233	19196,941		
	4	10655,287	16325,228			
	5	11732,578				



Proveedores con Giga-Ethernet.

La combinación que más se repite es proveedores Java, canal de eventos Java™ y consumidores Mono y además es la que tiene la mejor media. La peor combinación es un 61% más lenta que la mejor, el canal de eventos Mono es en media un 52% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	13340,976
2	Mono	Java	Mono	13402,981
3	Java	Java	Java	13524,549
4	Mono	Java	Java	13936,503
5	Java	Mono	Mono	19744,479
6	Mono	Mono	Mono	19772,282
7	Java	Mono	Java	21506,139
8	Mono	Mono	Java	21548,591



Comparativa de proveedores con Giga-Ethernet.

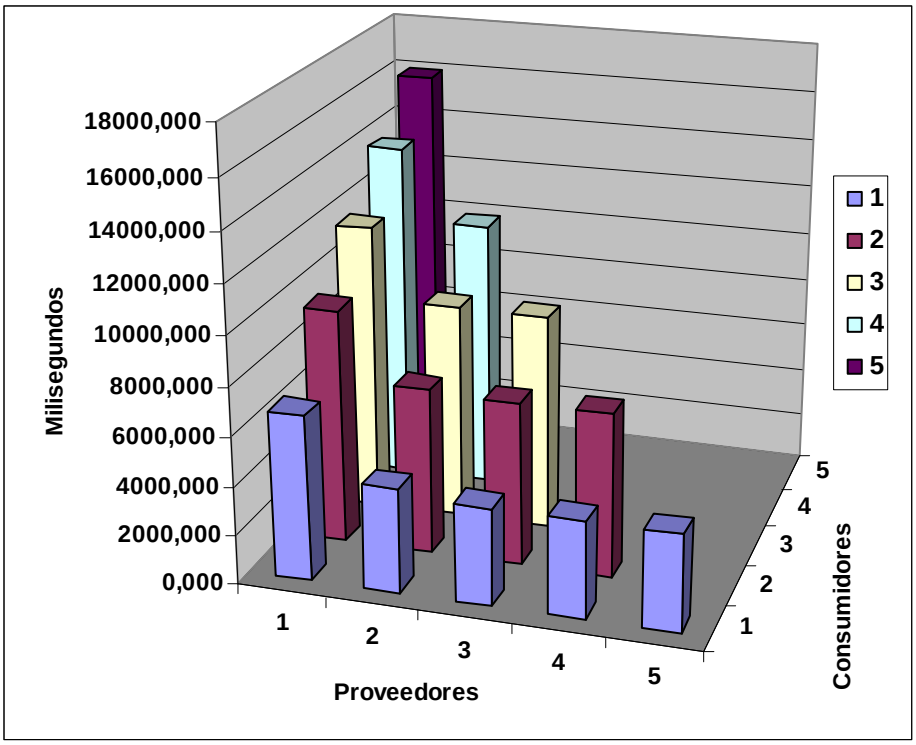
5.6.3. Consumidores con Fast-Ethernet.

Tipo prueba	Ancho de banda
Variable	Consumidor
Número eventos	1000
Interfaz	Fast-Ethernet
Unidad de tiempos	Milisegundos

Color		
Proveedores	Java	Mono
Canal Eventos	Java	Java
Consumidores	Mono	Mono

Mejores		Consumidores				
		1	2	3	4	5
Proveedores	1					
	2					
	3					
	4					
	5					

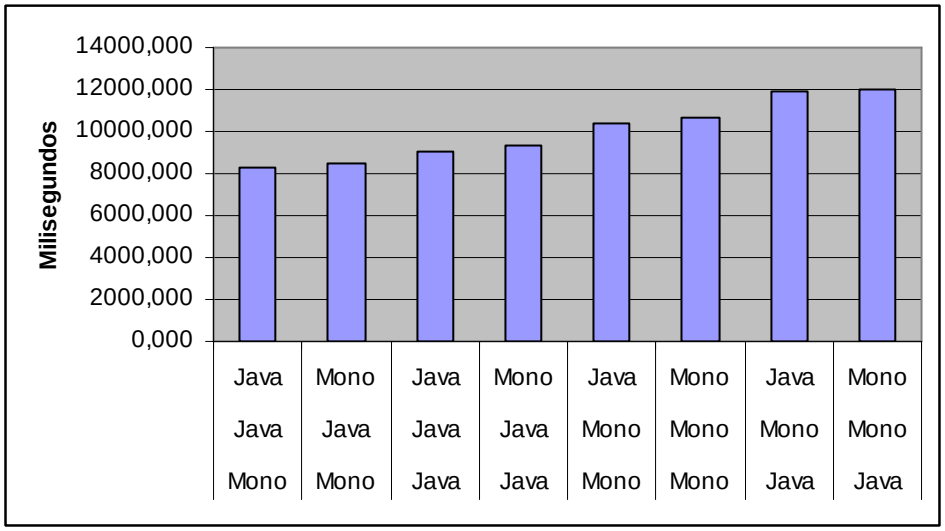
Media		Consumidores				
		1	2	3	4	5
Proveedores	1	6778,786	9612,802	11764,664	13965,497	16014,933
	2	4267,071	6794,507	8839,472	11038,169	
	3	3969,681	6707,027	8847,807		
	4	3955,730	6731,273			
	5	3949,780				



Consumidores con Fast-Ethernet.

La combinación que más se repite es proveedores Java, canal de eventos Java™ y consumidores Mono y además es la que tiene un mejor rendimiento. La peor combinación es un 61% más lenta que la mejor, el canal de eventos Mono es en media un 46% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	8252,316
2	Mono	Java	Mono	8448,695
3	Java	Java	Java	9032,465
4	Mono	Java	Java	9304,352
5	Java	Mono	Mono	10352,546
6	Mono	Mono	Mono	10651,400
7	Java	Mono	Java	11951,244
8	Mono	Mono	Java	12009,382



Comparativa de consumidores con Fast-Ethernet.

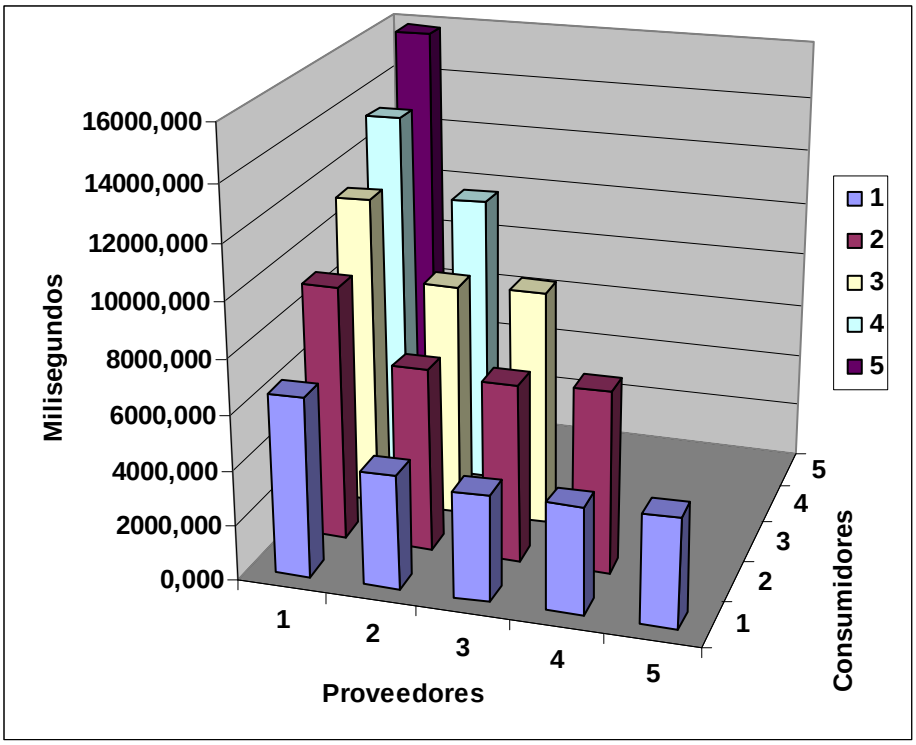
5.6.4. Consumidores con Giga-Ethernet.

Tipo prueba	Ancho de banda
Variable	Consumidor
Número eventos	1000
Interfaz	Giga-Ethernet
Unidad de tiempos	Milisegundos

Color		
Proveedores	Java	Mono
Canal Eventos	Java	Java
Consumidores	Mono	Mono

Mejores		Consumidores				
		1	2	3	4	5
Proveedores	1					
	2					
	3					
	4					
	5					

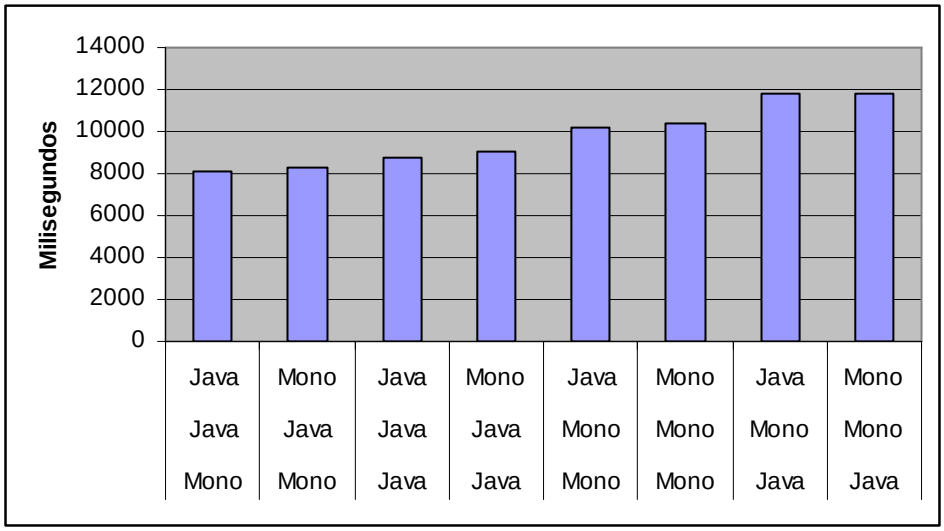
Media		Consumidores				
		1	2	3	4	5
Proveedores	1	6567,926	9321,354	11415,646	13514,319	15854,232
	2	4173,132	6662,259	8576,133	10685,203	
	3	3873,785	6522,083	8679,303		
	4	3895,825	6647,926			
	5	3948,851				



Consumidores con Giga-Ethernet.

La combinación que más se repite es proveedores Java, canal de eventos Java™ y consumidores Mono y además es la que tiene la media más rápida. La peor combinación es un 46% más lenta que la mejor, el canal de eventos Mono es en media un 29% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	8050,105
2	Mono	Java	Mono	8275,595
3	Java	Java	Java	8807,036
4	Mono	Java	Java	9064,403
5	Java	Mono	Mono	10176,126
6	Mono	Mono	Mono	10372,777
7	Java	Mono	Java	11766,324
8	Mono	Mono	Java	11813,514



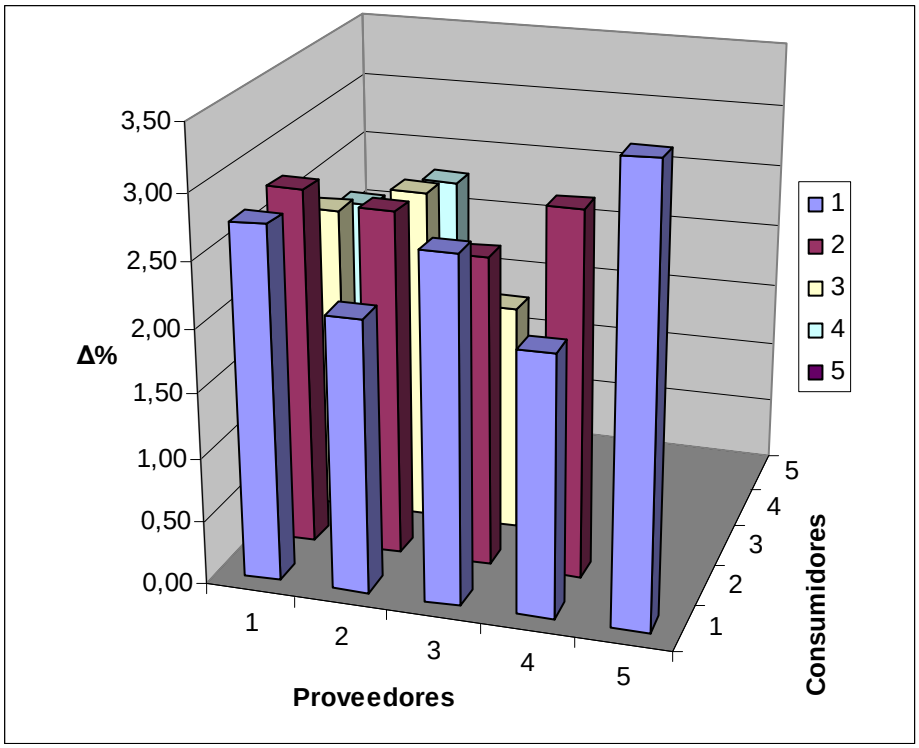
Comparativita de consumidores con Giga-Ethernet.

5.6.5. Comparativa de proveedores entre Fast/Giga Ethernet.

Tipo prueba	Ancho de banda
Número eventos	1000

Comparativa Fast/Giga Ethernet	
Variable	Proveedor
Salida	Diferencia % (Δ%)
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Fast}}{\overline{Giga}} - 1 \right) * 100$

Diferencia porcentual		Consumidores				
		1	2	3	4	5
Proveedores	1	2,75	2,80	2,43	2,26	1,40
	2	2,12	2,70	2,62	2,50	
	3	2,67	2,42	1,79		
	4	2,03	2,84			
	5	3,47				



Comparativa con proveedores de Fast/Giga Ethernet.

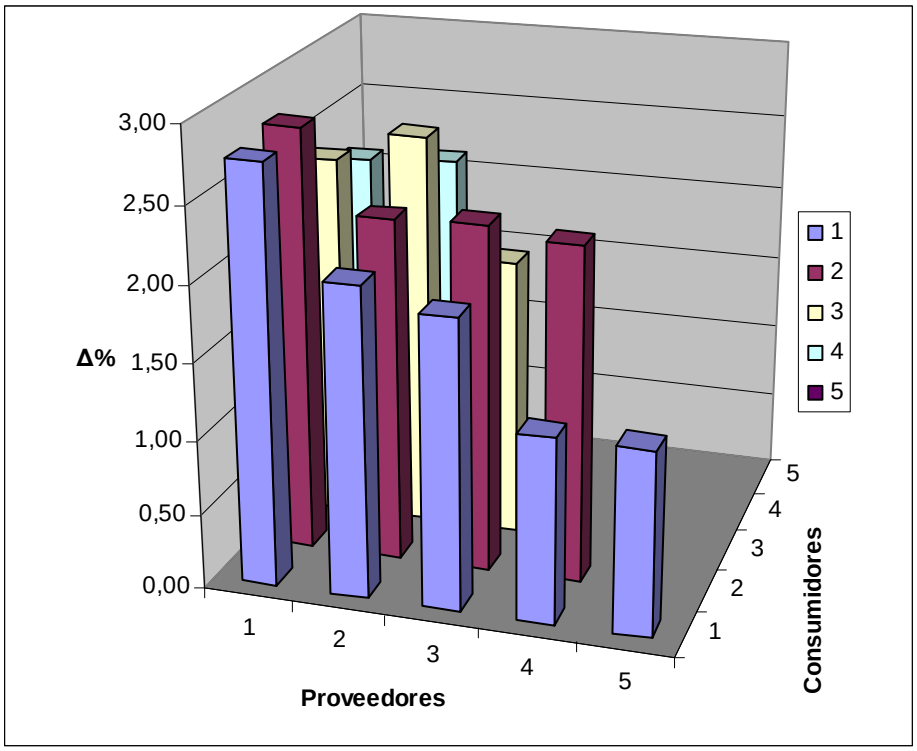
Como se puede ver en esta gráfica la merma de rendimiento de una red Fast-Ethernet con respecto a una red Giga-Ethernet en los proveedores no es muy significativa, una disminución media de 2,45%.

5.6.6. Comparativa de consumidores entre Fast/Giga Ethernet.

Tipo prueba	Ancho de banda
Número eventos	1000

Comparativa Fast/Giga Ethernet	
Variable	Consumidor
Salida	Diferencia % (Δ%)
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Fast}}{\overline{Giga}} - 1 \right) * 100$

Diferencia porcentual		Consumidores				
		1	2	3	4	5
Proveedores	1	2,75	2,80	2,43	2,26	1,40
	2	2,05	2,27	2,63	2,30	
	3	1,91	2,29	1,85		
	4	1,22	2,22			
	5	1,22				



Comparativa con consumidores de Fast/Giga Ethernet.

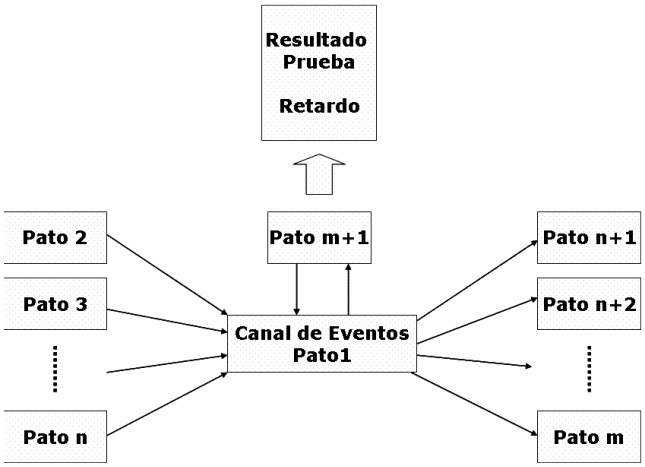
Como se puede ver en esta gráfica la merma de rendimiento de una red Fast-Ethernet con respecto a una red Giga-Ethernet en los consumidores, al igual que con los proveedores, no es muy significativa. La disminución media es de 2,10%.

5.7. Evaluación de la latencia.

La latencia es el lapso necesario para que un paquete de información viaje desde la fuente hasta su destino.

En esta prueba vamos a tratar de medir el tiempo que tarda en ir y volver un evento desde un cliente CORBA™, que denominaremos sonda, que actúa a la vez como proveedor y consumidor a través un canal de eventos. Al canal conectaremos más proveedores y consumidores, cada uno de esos clientes CORBA™ se encargará de un único rol.

Cada vez que ejecutemos una prueba enviaremos y recibiremos 10 eventos. Se guardará en un archivo el tiempo de ir y volver a través del canal de cada evento. El resultado de esa prueba será la media de los 5 valores más pequeños medidos, la máxima dispersión de dichos dichas medidas será inferior al 20%.



Evaluación de la latencia.

Cada evento contendrá un entero, los enteros ocupan 4 *bytes*. Los enteros enviados por la sonda para distinguirlos del resto serán números negativos.

En el CD que acompaña esta documentación se incluyen todos los resultados obtenidos, en esta memoria nos vamos a quedar con los mejores tiempos.

Como es natural una solución es mejor que otra cuando el tiempo necesario para ir y volver de la sonda al canal sea menor.

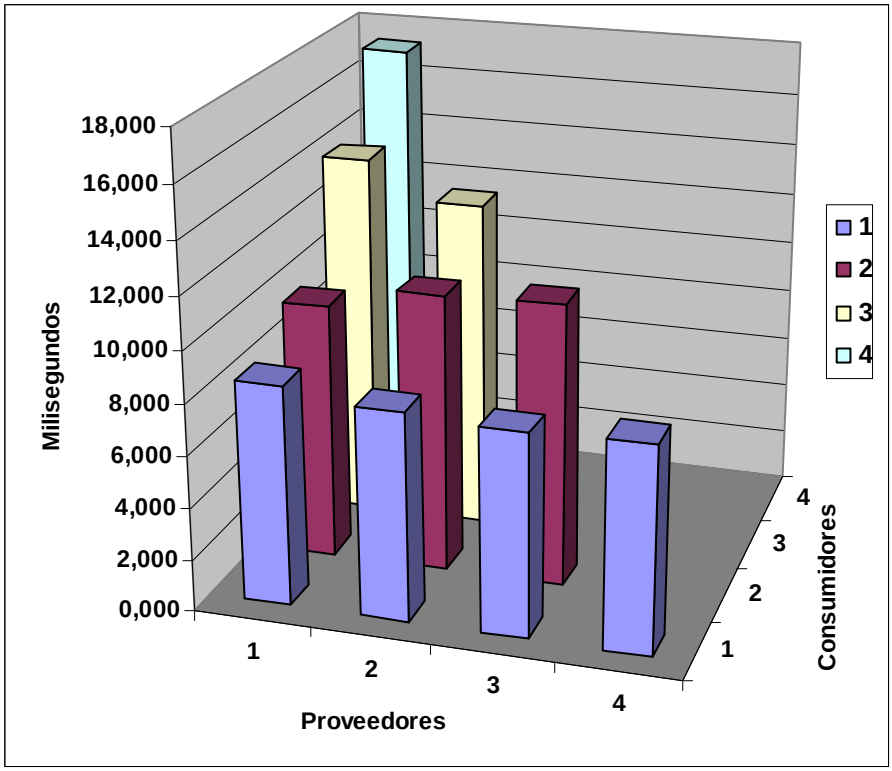
5.7.1. Sonda Java™ y red Fast-Ethernet.

Tipo prueba	Latencia
Interfaz	Fast-Ethernet
Sonda	Java
Unidad de tiempos	Milisegundos

Color				
Proveedores	Java	Java	Mono	Mono
Canal Eventos	Java	Java	Java	Mono
Consumidores	Java	Mono	Mono	Mono

Mejores		Consumidores			
		1	2	3	4
Proveedores	1				
	2				
	3				
	4				

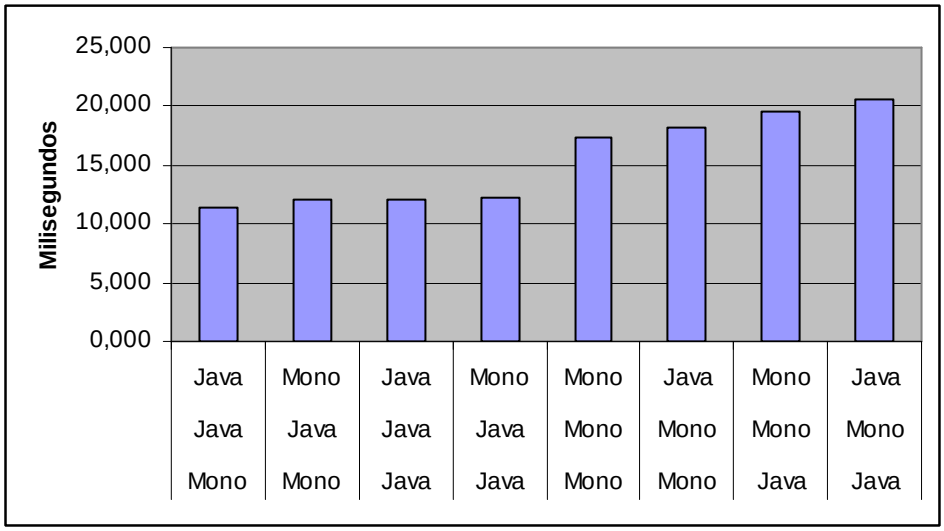
Media		Consumidores			
		1	2	3	4
Proveedores	1	8,531	9,921	14,238	17,339
	2	8,069	10,800	12,822	
	3	7,859	10,927		
	4	7,988			



Sonda Java™ y red Fast-Ethernet.

La combinación que más se repite es proveedores Java, canal de eventos Java™ y consumidores Mono y además es la que tiene la media más rápida. La peor combinación, que tiene proveedores y canal de eventos Mono y consumidores Java™, es un 80% más lenta que la mejor. El canal de eventos Mono es en media un 58% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	11,453
2	Mono	Java	Mono	12,018
3	Java	Java	Java	12,093
4	Mono	Java	Java	12,307
5	Mono	Mono	Mono	17,396
6	Java	Mono	Mono	18,132
7	Mono	Mono	Java	19,641
8	Java	Mono	Java	20,613



Comparativa sonda Java™ y red Fast-Ethernet.

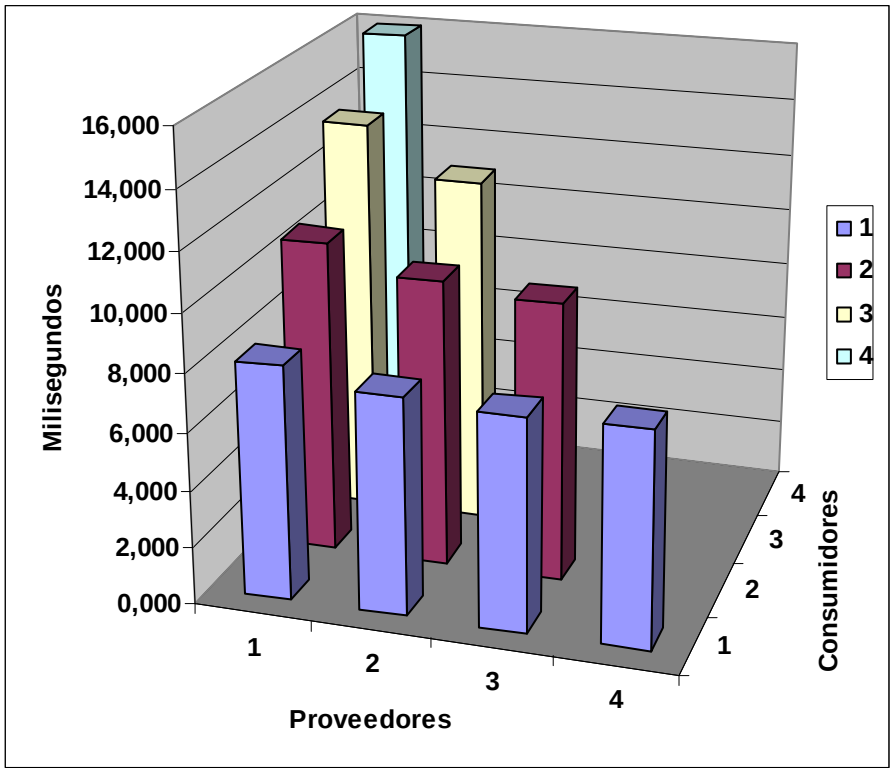
5.7.2. Sonda Mono y red Fast-Ethernet.

Tipo prueba	Latencia
Interfaz	Fast-Ethernet
Sonda	Mono
Unidad de tiempos	Milisegundos

Color			
Proveedores	Java	Java	Mono
Canal Eventos	Java	Java	Java
Consumidores	Java	Mono	Mono

Mejores		Consumidores			
		1	2	3	4
Proveedores	1				
	2				
	3				
	4				

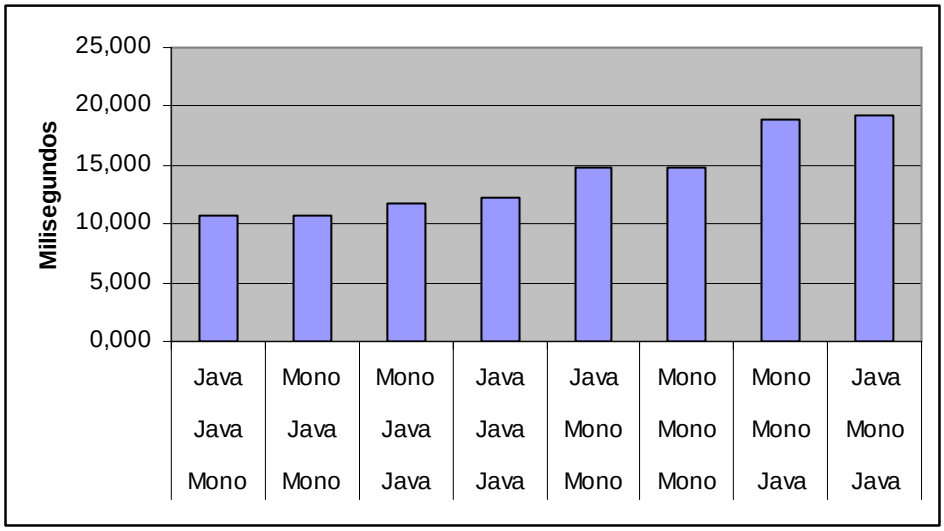
Media		Consumidores			
		1	2	3	4
Proveedores	1	8,132	10,916	13,808	15,971
	2	7,555	10,007	12,160	
	3	7,366	9,678		
	4	7,434			



Sonda Mono™ y red Fast-Ethernet.

Como podemos ver en el gráfico las combinaciones que más se repiten son proveedores Mono, canal de eventos Java™ y consumidores Mono y proveedores Java, canal de eventos Java™ y consumidores Mono con 6 veces cada una. A continuación en el ranking podremos ver que la combinación más rápida es proveedores Java, canal de eventos Java™ y consumidores Mono. La peor combinación, con proveedores y consumidores Java™ y canal de eventos Mono, es un 81% más lenta que la mejor. El canal de eventos Mono es en media un 49% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Java	Java	Mono	10,633
2	Mono	Java	Mono	10,719
3	Mono	Java	Java	11,668
4	Java	Java	Java	12,223
5	Java	Mono	Mono	14,750
6	Mono	Mono	Mono	14,879
7	Mono	Mono	Java	18,843
8	Java	Mono	Java	19,286



Comparativa de sonda Java™ y red Fast-Ethernet.

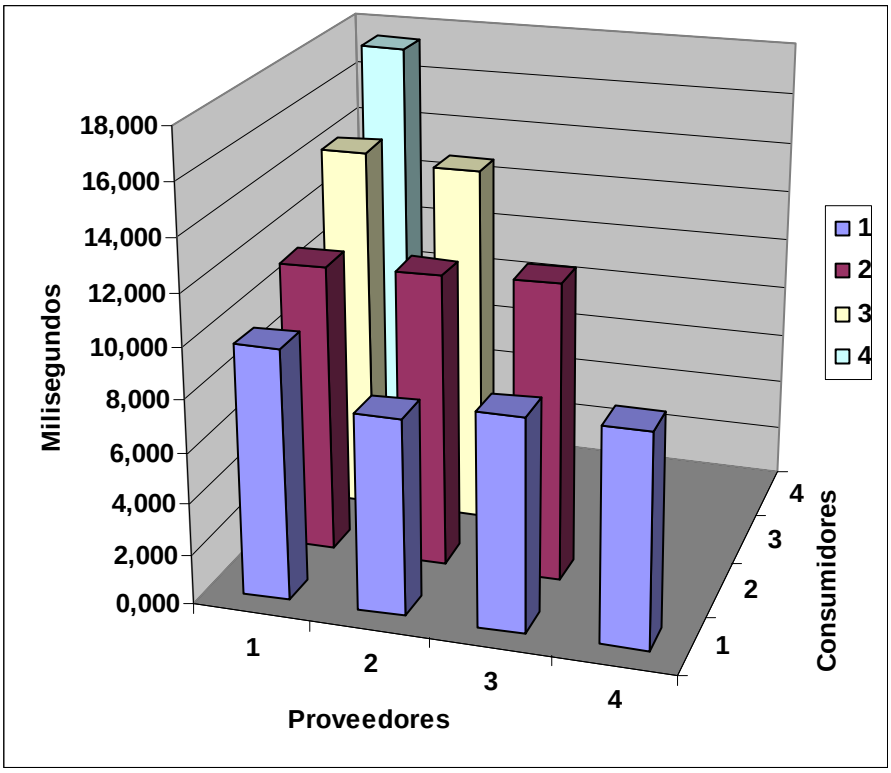
5.7.3. Sonda Java™ y red Giga-Ethernet.

Tipo prueba	Latencia
Interfaz	Giga-Ethernet
Sonda	Java
Unidad de tiempos	Milisegundos

Color		
Proveedores	Java	Mono
Canal Eventos	Java	Java
Consumidores	Mono	Mono

Mejores		Consumidores			
		1	2	3	4
Proveedores	1				
	2				
	3				
	4				

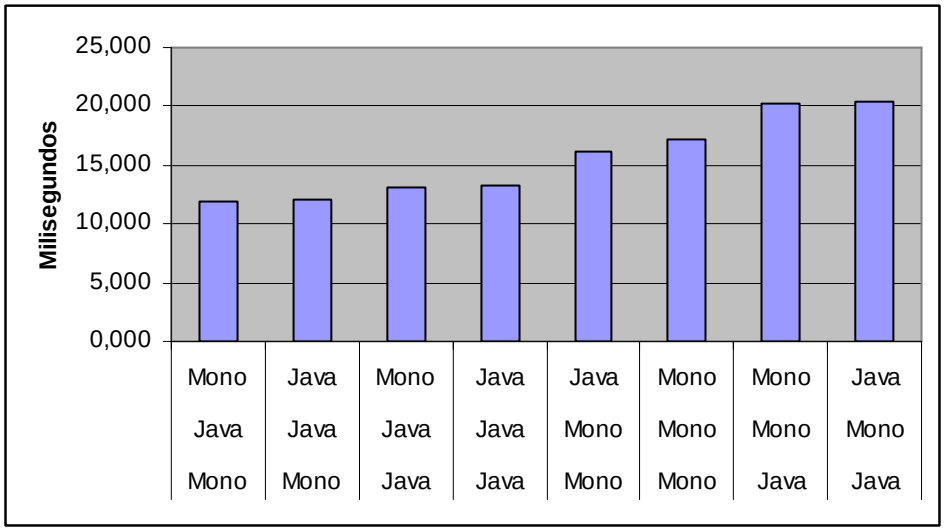
Media		Consumidores			
		1	2	3	4
Proveedores	1	9,799	11,349	14,459	17,415
	2	7,685	11,474	14,094	
	3	8,297	11,616		
	4	8,312			



Sonda Java™ y red Giga-Ethernet.

La combinación que más se repite es proveedores Mono, canal de eventos Java™ y consumidores Mono y además es la que tiene la media más rápida. La peor combinación, con proveedores y consumidores Java™ y canal de eventos Mono, es un 71% más lenta que la mejor. El canal de eventos Mono es en media un 47% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Mono	Java	Mono	11,888
2	Java	Java	Mono	12,062
3	Mono	Java	Java	13,123
4	Java	Java	Java	13,235
5	Java	Mono	Mono	16,227
6	Mono	Mono	Mono	17,240
7	Mono	Mono	Java	20,255
8	Java	Mono	Java	20,327



Comparativa sonda Java™ y red Fast-Ethernet.

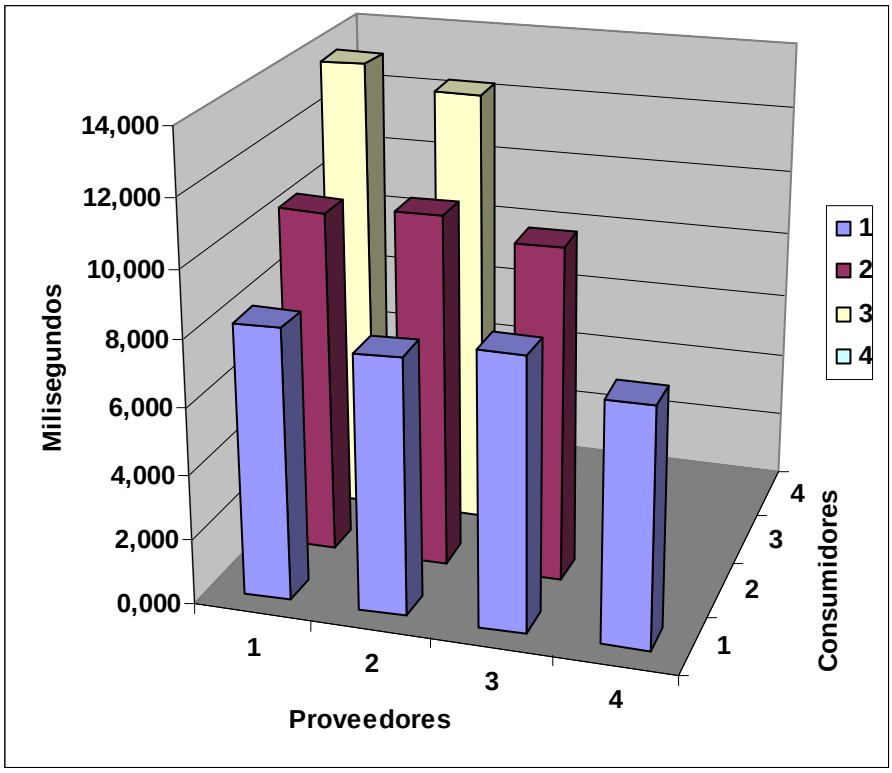
5.7.4. Sonda Mono y red Giga-Ethernet.

Tipo prueba	Latencia
Interfaz	Giga-Ethernet
Sonda	Mono
Unidad de tiempos	Milisegundos

Color			
Proveedores	Java	Java	Mono
Canal Eventos	Java	Java	Java
Consumidores	Java	Mono	Mono

Mejores		Consumidores			
		1	2	3	4
Proveedores	1				
	2				
	3				
	4				

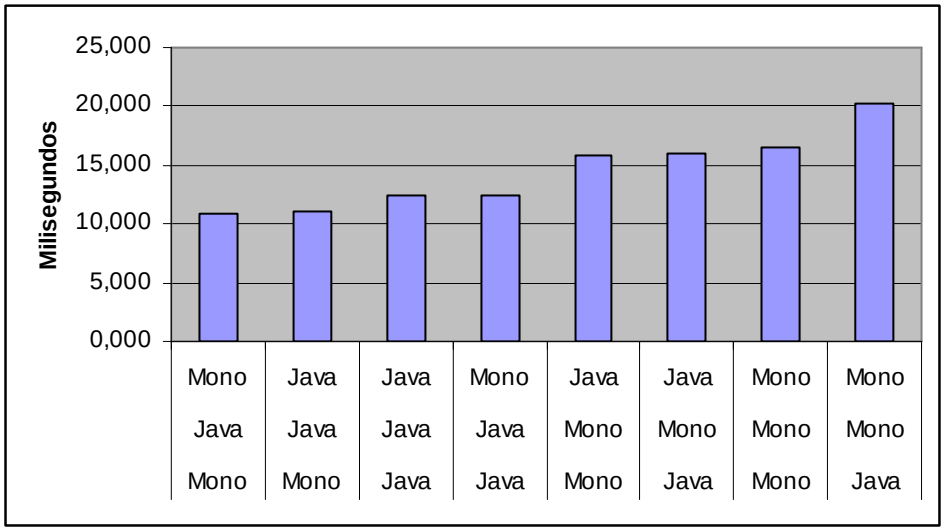
Media		Consumidores			
		1	2	3	4
Proveedores	1	8,226	10,454	13,924	0,017
	2	7,745	10,671	13,274	
	3	8,227	10,106		
	4	7,217			



Sonda Mono y red Giga-Ethernet.

La combinación que más se repite es proveedores Mono, canal de eventos Java™ y consumidores Mono y además es la que tiene la media más rápida. La peor combinación, con proveedores y canal de eventos Mono y consumidores Java™, es un 85% más lenta que la mejor. El canal de eventos Mono es en media un 47% más lento que el canal de eventos Java™:

	Proveedores	Canal de eventos	Consumidores	Media
1	Mono	Java	Mono	10,946
2	Java	Java	Mono	11,115
3	Java	Java	Java	12,386
4	Mono	Java	Java	12,407
5	Java	Mono	Mono	15,855
6	Java	Mono	Java	15,943
7	Mono	Mono	Mono	16,580
8	Mono	Mono	Java	20,268



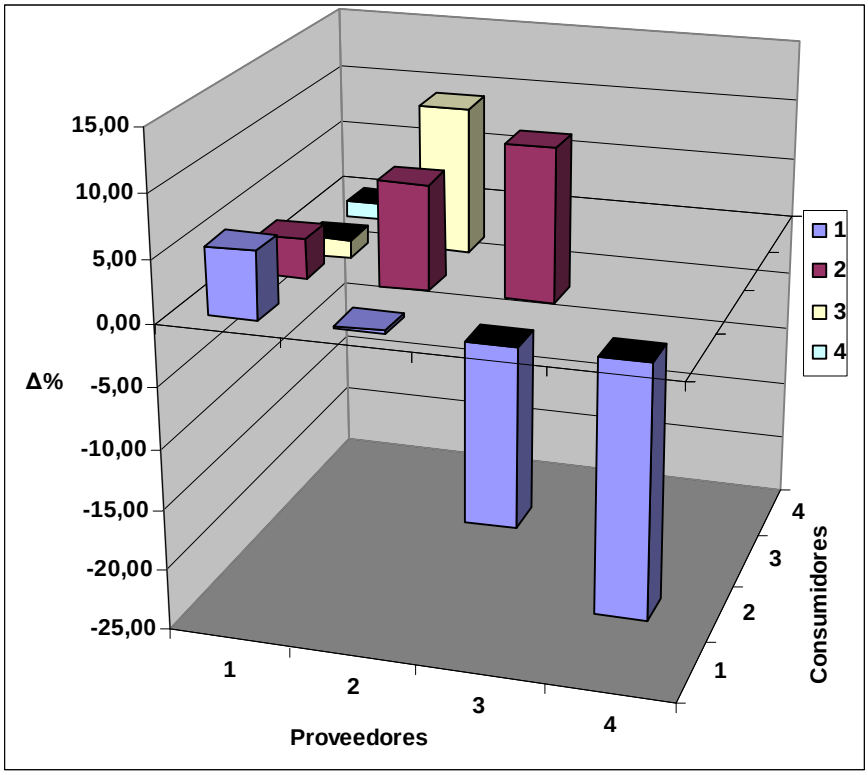
Comparativa sonda Mono y red Giga-Ethernet.

5.7.5. Comparativa entre Fast/Giga Ethernet con sonda Java™.

Tipo prueba	Retardo

Comparativa Fast/Giga Ethernet	
Sonda	**Java**
Salida	**Diferencia % (Δ%)**
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Fast}}{\overline{Giga}} - 1 \right) * 100$

Incremento porcentual		Consumidores			
		1	2	3	4
Proveedores	1	5,42	3,24	-1,49	-1,25
	2	0,24	8,43	11,74	
	3	-14,16	12,12		
	4	-20,32			



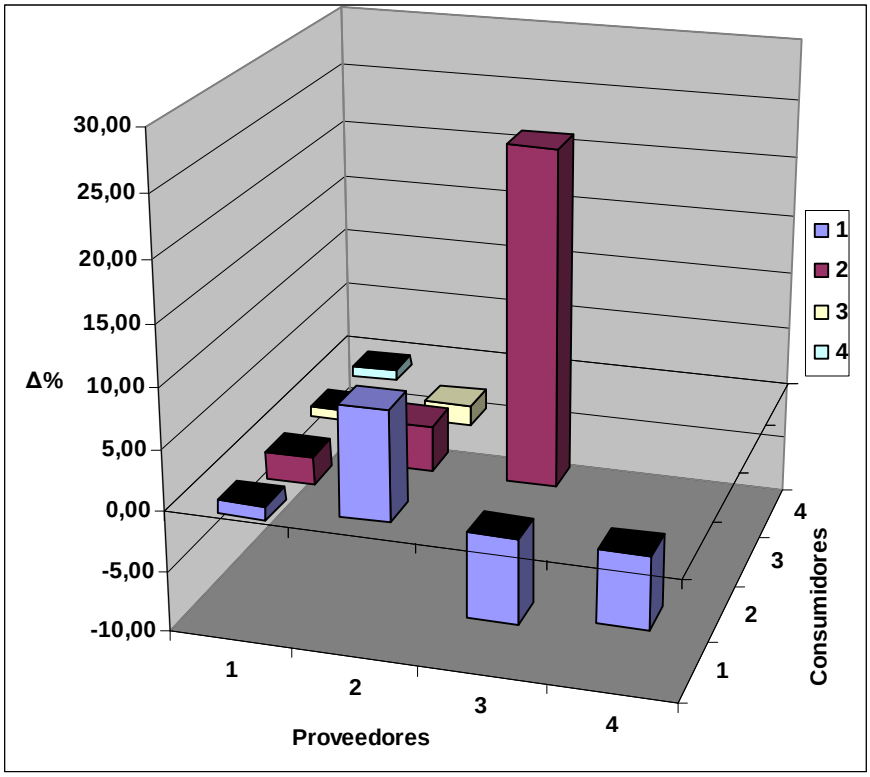
Comparativa red Fast/Giga-Ethernet y sonda Java™.

Como se puede ver en esta gráfica la merma de rendimiento de una red Fast-Ethernet con respecto a una red Giga-Ethernet con una sonda Java™ es despreciable, incluso hay casos dónde la red Fast-Ethernet ha sido mejor. La disminución media es de 0,4%.

5.7.6. Comparativa entre Fast/Giga Ethernet con sonda Mono.

Comparativa Fast/Giga Ethernet	
Sonda	**Mono**
Salida	**Diferencia % (Δ%)**
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Fast}}{\overline{Giga}} - 1 \right) * 100$

Incremento porcentual		**Consumidores**			
		1	2	3	4
Proveedores	1	-1,11	-2,39	-0,84	-0,84
	2	9,04	3,82	1,65	
	3	-7,05	27,06		
	4	-5,93			



Comparativia red Fast/Giga-Ethernet y sonda Mono.

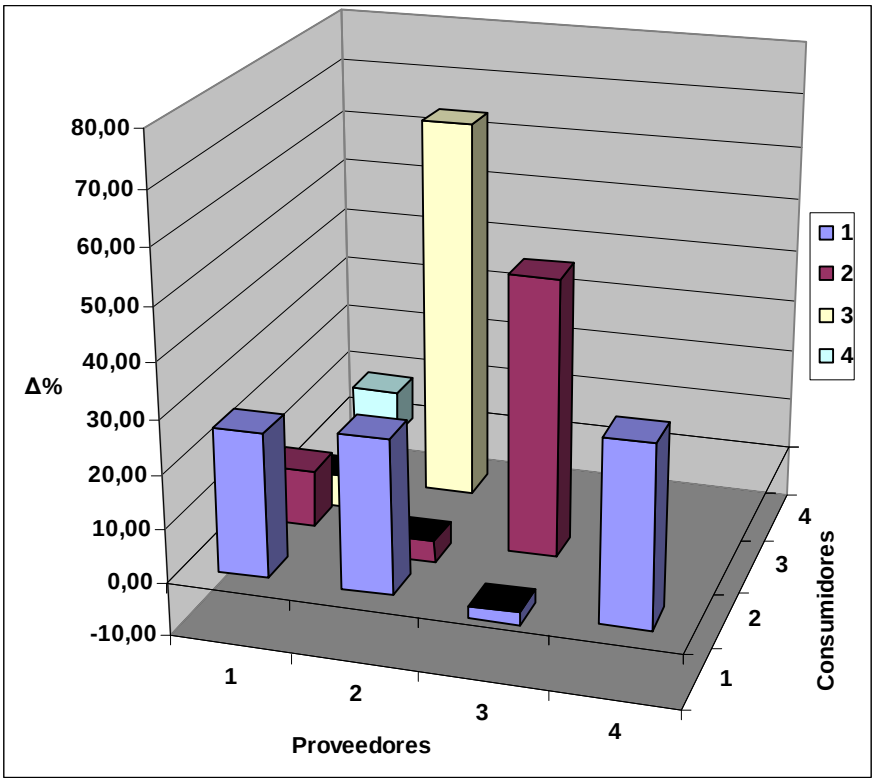
Como se puede ver en esta gráfica la merma de rendimiento de una red Fast-Ethernet con respecto a una red Giga-Ethernet con una sonda Java™ no es muy significativa, existiendo casos donde la red Fast-Ethernet ha sido mejor. La disminución media es de 2,34%.

5.7.7. Comparativa entre Sondas Mono/Java con Fast-Ethernet.

Tipo prueba	Retardo

Comparativa Sonda Mono/Java	
Interfaz	**Fast-Ethernet**
Salida	**Diferencia % (Δ%)**
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Mono}}{\overline{Java}} - 1 \right) * 100$

Incremento porcentual		Consumidores			
		1	2	3	4
Proveedores	1	26,79	10,56	-6,38	8,62
	2	28,36	-4,23	70,06	
	3	-2,45	51,10		
	4	33,19			



Comparativas entre sondas Mono/Java con Fast-Ethernet.

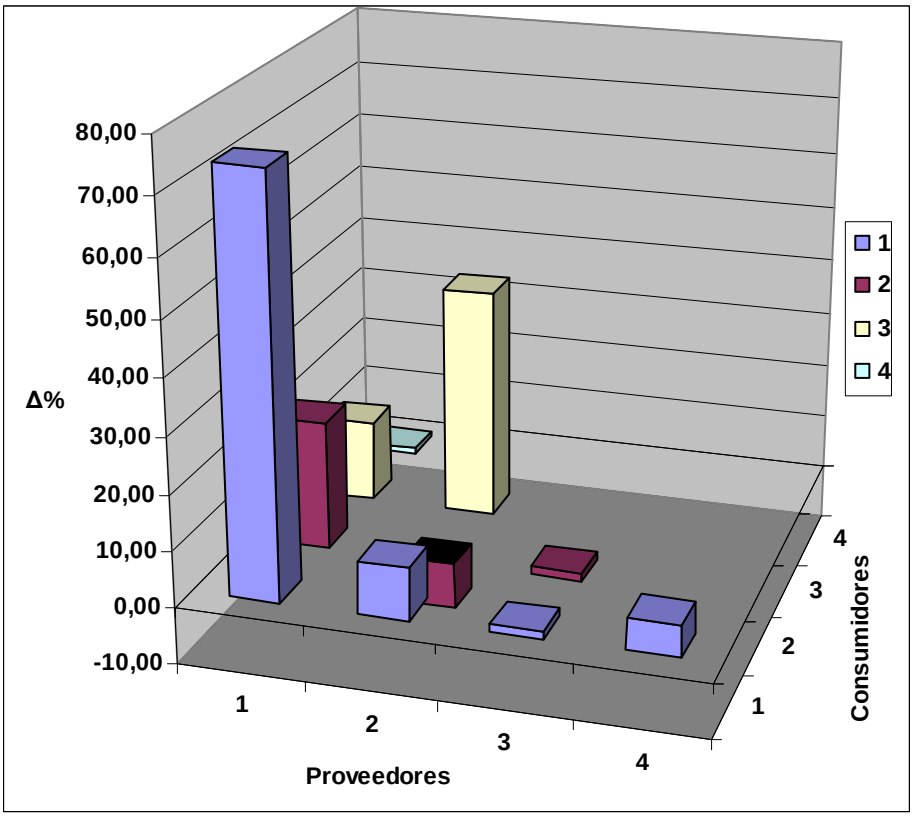
Como se puede ver en esta gráfica la merma de rendimiento de una sonda Mono con respecto a una sonda Java™ con una red Fast-Ethernet es significativa con una disminución media de 21,56%. Aún así hay casos donde la sonda Mono ha sido mejor.

5.7.8. Comparativa entre Sondas Mono/Java con Giga-Ethernet.

Tipo prueba	Retardo

Comparativa Sonda Mono/Java	
Interfaz	**Giga-Ethernet**
Salida	**Diferencia % (Δ%)**
Cálculo de la salida	$\Delta\% = \left(\frac{\overline{Mono}}{\overline{Java}} - 1 \right) * 100$

Incremento porcentual		Consumidores			
		1	2	3	4
Proveedores	1	74,17	23,13	14,36	0,96
	2	9,41	-8,22	40,99	
	3	1,31	1,46		
	4	5,73			



Comparativas entre sondas Mono/Java con Giga-Ethernet.

Como se puede ver en esta gráfica la merma de rendimiento de una sonda Mono con respecto a una sonda Java™ con una red Fast-Ethernet no es un poco significativa con una disminución media de 16,33%. Aún así hay casos donde la sonda Mono ha sido mejor.

5.8. Conclusiones.

En todos los *rankings* hechos en las sucesivas pruebas se puede ver que el canal de eventos Mono es en media un 50% más lento que la implementación en Java™.

Posteriormente se puede observar que con el mismo canal de eventos una configuración donde los consumidores de eventos estén implementados en Mono es más rápida que cualquier otra dónde los consumidores son Java™. Las combinaciones con consumidores tipo Mono son un 9,94 mejores que aquellas que tienen consumidores Java™

En el tercer factor a tener en cuenta, los proveedores, hay más igualdad en los resultados, aunque la configuración dónde los proveedores son Java™ normalmente es algo más rápida que aquella que teniendo el canal de eventos y los consumidores del mismo tipo, varía en los proveedores, siendo estos naturalmente del tipo Mono. La diferencia media entre las configuraciones con proveedores Java™ y las que tienen proveedores de tipo Mono es de 1,56% a favor de las primeras.

El canal de eventos es el factor más determinante según hemos podido ver en el rendimiento de una aplicación de este tipo, más aún que la

propia red donde solamente se puede ganar alrededor de un 3% usando Giga-Ethernet en lugar de Fast-Ethernet, según podemos ver en las pruebas de ancho de banda.

La diferencia de rendimiento con las pruebas de latencia entre la red Giga-Ethernet y la Fast-Ethernet varía mucho, en algunos casos sucede que Fast-Ethernet es la mejor solución. En las comparaciones de las pruebas de retardo cuando se tienen en cuenta las medias de todas las combinaciones, como se ha hecho en este estudio, es normal que sucedan estas cosas. En estos casos hay que tener en cuenta que el estudio de la latencia no puede ser tan preciso como con las pruebas de ancho de banda al tratarse de unos tiempos mucho más pequeños.

De lo observado aquí la principal conclusión que se puede obtener es que el canal de eventos, el elemento de los tres que componen la infraestructura del servicio de eventos, que además es aquel que menos depende de la aplicación que deseemos realizar con este servicio, es muy conveniente que sea Java™ y no Mono.

Los proveedores y consumidores no son causantes de un impacto tan importantes en el rendimiento de la aplicación para que tengamos el plantearnos llegar a cambiar la plataforma en que los desarrollaremos. El desarrollador podrá usar aquella plataforma de desarrollo que más se ajuste a sus capacidades o a la naturaleza del problema.

6. Desarrollo de herramienta de monitorización de un Cluster.

6.1. Descripción del problema.

Disponemos de un cluster de 8 máquinas sobre el que se ejecutan aplicaciones paralelas y necesitamos una herramienta que nos muestre los porcentajes de memoria y CPU en cada una de las máquinas a intervalos de tiempo con un interfaz gráfico. Cada porcentaje obtenido será un valor entero entre 0 y 100.

Todas las máquinas del cluster usan el sistema operativo Linux® y tienen instalado el JDK5.0™ y la máquina virtual de Mono v1.9.

Solamente una de las máquinas del cluster tiene acceso por red al exterior. El administrador del sistema solamente permite a los usuarios el acceso al cluster a través de red por lo que estos se conectan a la máquina con conexión externa y de ahí acceden al resto. A los usuarios se les permite tener servidores TCP/IP escuchado por encima del puerto 1024.

Los usuarios han expresado su deseo de disponer de interfaces de usuario gráfico tanto en Windows como en Linux®. Además han explicado que es posible que varios usuarios estén usando uno de estos interfaces a la vez desde distintas máquinas fuera del cluster.

6.2. Solución propuesta.

Durante el apartado 3 de esta documentación se ha analizado varías soluciones de interoperatividad entre plataforma CLI y Java™. Con esta aplicación además de solucionar el problema también pretendemos demostrar la viabilidad de las mismas en un problema real.

En la descripción del problema se nos presenta una aplicación que tiene 8 entradas de información con el porcentaje de CPU y memoria desde cada uno de los equipos que componen el cluster; puede tener varías salidas de información a través de cada uno de los interfaces gráficos usados por los usuarios desde sus máquinas fuera del cluster y donde además solamente una máquina tiene acceso directo al resto, es decir, a las máquinas de dentro y fuera del cluster.

De entre las soluciones planteadas en el apartado 3 vamos a escoger el canal de eventos CORBA™ con el modelo de inyección. Todas las máquinas tienen instalado Java™ y Mono, así que no hay ningún problema en implementar la solución con Java™ y C#

El canal de eventos CORBA™ es una solución ideada para comunicar varios proveedores de información con múltiples consumidores de información. En esta solución solamente es necesario que cada uno de los consumidores y cada unos de proveedores sean capaces de conectar con una única máquina donde se hospeda el canal de eventos.

El canal de eventos sigue el modelo de comunicación a través de mensajes de publicación/suscripción, según se comenta en el apartado

3.7.1. El modelo de publicación/suscripción no guarda información hasta que esta es consumida, esto no es necesario en este problema ya que lo que quieren los usuarios es simplemente disponer de los datos de uso de CPU y memoria más recientes. Un modelo de comunicación a través de mensajes punto a punto no es válido en este caso ya que la información tiene que ser enviada a todos los usuarios, no solamente a uno de cada vez como sucede con este modelo.

Hemos escogido el modelo de inyección para que el canal no se vea obligado a sondear continuamente a los proveedores de información de cada máquina como sucede en el modelo de extracción tanto con la implementación de Jacorb como en la realizada en este proyecto. El sondeo continuo por parte del canal de eventos causa un gasto de CPU alto que además de ralentizar mucho al resto de aplicaciones del cluster haría inservibles las medidas de porcentajes de uso de CPU.

Vamos a permitir que en los emisores de información donde se realiza la medida de CPU y memoria esta se haga con un periodo configurable. El periodo de medida debe de ser fijado de manera que altere lo menos posible el rendimiento del resto de aplicaciones del cluster y sea bastante pequeño para que las medidas se actualicen los suficiente.

6.3. Modelado de la solución con un canal de eventos.

En la arquitectura de un canal de eventos están presentes los proveedores de información, los consumidores de información, el propio canal de eventos y los eventos.

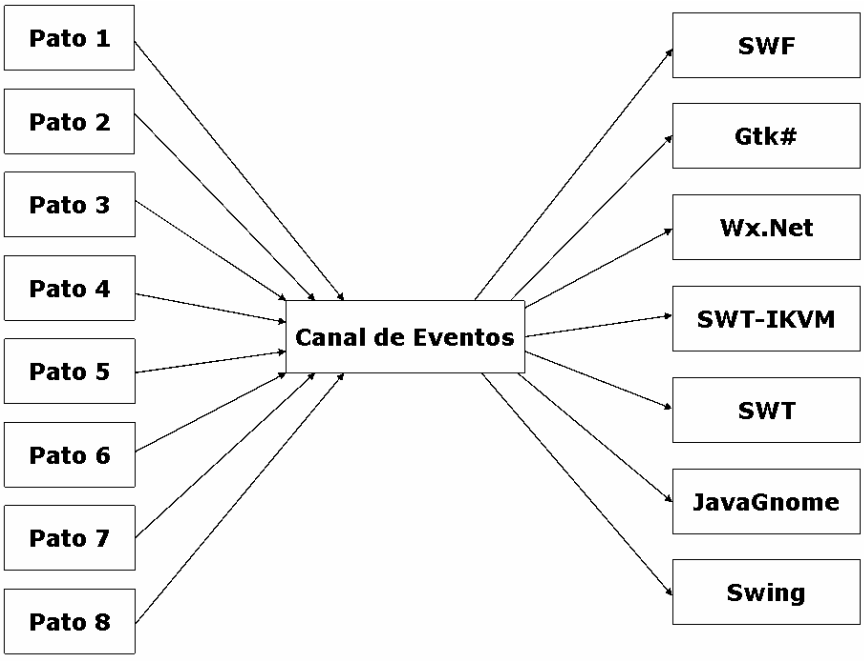
Para realizar una aplicación hemos de fijar cada una de las partes de la arquitectura, comenzaremos fijando proveedores, consumidores y canal de eventos.

Aquí hemos de extraer la información sobre CPU y Memoria desde cada máquina del cluster, por tanto los proveedores de eventos del canal serán los programas que lean los datos a observar y los envíen empaquetados en un evento al canal. Los proveedores pasarán los eventos al canal con un periodo fijado por parámetro al arrancar el proveedor.

El canal de eventos tiene que estar en una máquina que pueda comunicarse por red con todas las máquinas participantes, aquí solamente tenemos una que cumple con esa condición y es la máquina del cluster que tiene acceso por red al exterior.

Los consumidores de eventos extraen la información del canal. Aquí los programas que extraen información del canal son los interfaces gráficos de los usuario donde se muestra los porcentajes de CPU y memoria, así que podemos identificar a los interfaces gráficos como los consumidores del canal.

Una vez identificados los proveedores, los consumidores y el canal de eventos tenemos que la arquitectura de la solución quedaría:



Modelado del problema con el canal de eventos.

Los eventos es la manera que tiene el servicio de eventos de CORBA™ de denominar a la información que viaja por el canal. Los eventos o mensajes en toda solución MOM, *middleware* orientado a eventos, deben de ser auto contenidos, es decir, no deben de requerir ni contexto ni ningún tipo de información auxiliar externa al evento para ser interpretados por un consumidor o receptor de eventos. Debemos de introducir en el evento el porcentaje de CPU y memoria, como ya se comentó en la descripción del problema, y además de eso también hay que añadir un identificador de la máquina que lo envió. Como identificador de la máquina que envía el evento nos servirá un entero cuyos valores irán de 1 a 8.

El canal de eventos que hemos implementado solamente envía tipos básicos de CORBA™ dentro de un *Any*. Si quisiéramos que enviase tipos compuestos deberíamos de extender el canal de eventos genérico para convertirlo en uno con tipo.

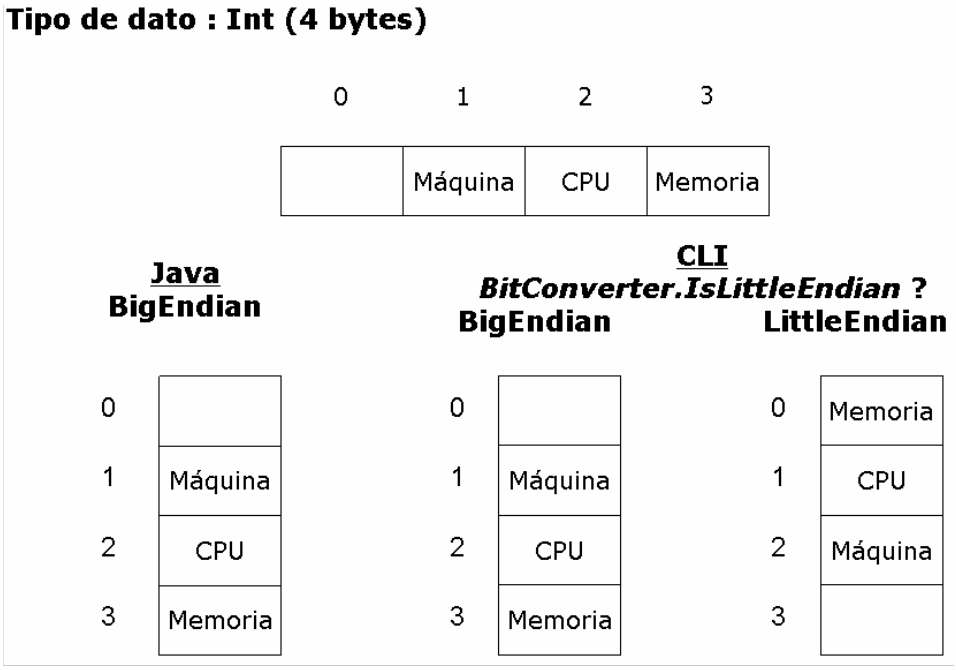
Antes hemos identificado que lo que tenemos que meter en cada evento son tres enteros, cuyos valores están entre 0 y 100 para los porcentajes de memoria y CPU y entre 1 y 8 para el identificador de máquina. Un *byte* es suficiente para enviar enteros cuyos valores oscilen entre 0 y 255, intervalo de valores más amplio que el que necesitamos para cada dato de los que enviamos por el canal.

Un *Int* ocupa cuatro *bytes* en Java, en C# y en CORBA™, es el tipo básico que vamos a usar para enviar los eventos. Tanto C# como Java™ incorporan clases con las que podemos pasar un entero a un *array* de cuatro *bytes*.

El procedimiento para enviar los datos que necesitamos en un tipo básico será introducir cada uno de los tres valores en posiciones de un *array* de cuatro *bytes* y convertirlo a un *Int* que si podemos enviar, el consumidor se encargará de hacer el procedimiento contrario.

En Java™ podemos asumir que el orden de los *bytes* en un *Int* es siempre *bigendian*. En C# el *endian* de los valores en un *Int* es el mismo del de la arquitectura donde está implementada la maquina virtual, para saber el *endian* debemos de consultar el valor de la propiedad *BitConverter.IsLittleEndian*.

Los eventos se interpretarán de la siguiente forma:



Valor de los eventos.

6.4. Adquisición de datos.

El sistema de ficheros *proc* de Linux® está compuesto por un conjunto de ficheros y directorios virtuales a través de los cuales podemos obtener información del núcleo del sistema operativo en funcionamiento. Este sistema de ficheros normalmente se monta en el directorio */proc* del sistema de ficheros raíz del sistema.

6.4.1. Uso de CPU.

El uso de CPU se lee del archivo */proc/loadavg*.

El archivo /proc/loadavg proporciona una visión sobre la utilización media del procesador.

Está compuesto de una sola línea, un ejemplo de fichero *loadavg* podría ser el siguiente:

```
0.20 0.18 0.12 1/80 11206
```

Las primeras tres columnas contienen medidas del uso de memoria en los últimos periodos de 1, 5 y 10 minutos. La cuarta columna contiene el número total de procesos en ejecución y el número total de procesos en el sistema. La última columna contienen el ID del último proceso usado.

El % de uso de CPU que enviamos es el resultado de pasar de tanto por uno a tanto por cien la primera columna del fichero.

6.4.2. Uso de Memoria.

El uso de memoria se lee del archivo */proc/meminfo*.

El archivo */proc/meminfo* proporciona información de una serie de variables relativas al uso de memoria RAM en el procesador.

Todas las líneas de */proc/meminfo* a excepción de la primera comienzan por el nombre de la variable seguido de dos puntos y el valor de la misma.

Un ejemplo de archivo */proc/meminfo* es el siguiente:

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  261709824 253407232  8302592        0 120745984 48689152
Swap: 402997248     8192 402989056
MemTotal:       255576 kB
MemFree:          8108 kB
MemShared:           0 kB
Buffers:        117916 kB
Cached:          47548 kB
Active:         135300 kB
Inact_dirty:     29276 kB
Inact_clean:       888 kB
Inact_target:        0 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       255576 kB
LowFree:          8108 kB
SwapTotal:      393552 kB
SwapFree:       393544 kB
```

Según la versión de Linux® el orden en que se muestran los valores de las distintas medidas puede variar.

Para calcular el valor del tanto por ciento de memoria usada leeremos de este fichero la cantidad de memoria total, MemTotal, y la cantidad de memoria libre, MemFree.

La fórmula usada para calcular el porcentaje de memoria usada es la siguiente:

```
((MemTotal - MemFree) / MemTotal) * 100);
```

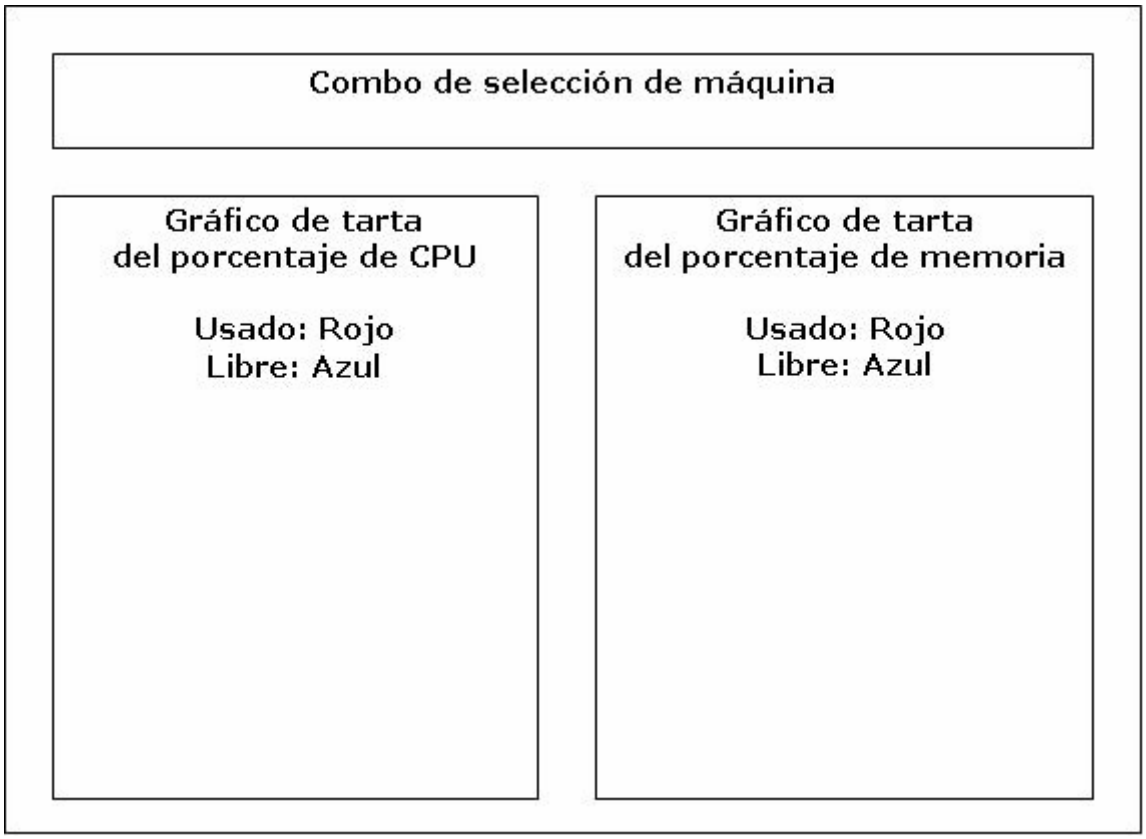
6.4. Interfaz de usuario.

En la descripción del problema se solicita que los datos de porcentaje de CPU y memoria deben de mostrarse en un interfaz gráfico. Tanto en Java™ como en Mono y en .Net existen variedad de *toolkits* gráficos disponibles en las plataformas solicitadas por los usuarios, Linux® y Windows®. Partiendo un prototipo se han desarrollado varios clientes

usando aquellos *toolkits* disponibles en las distintas plataformas que cumplieran los siguientes requisitos:

- Estar disponible en Java, Mono o .Net
- Tener una implementación funcional en Windows y/o Linux®.
- No tener una licencia "vírica" que nos impida distribuir nuestro programa de la manera que estimemos oportuna.
- No tratarse de una implementación cuyos autores han dejado de mantener.

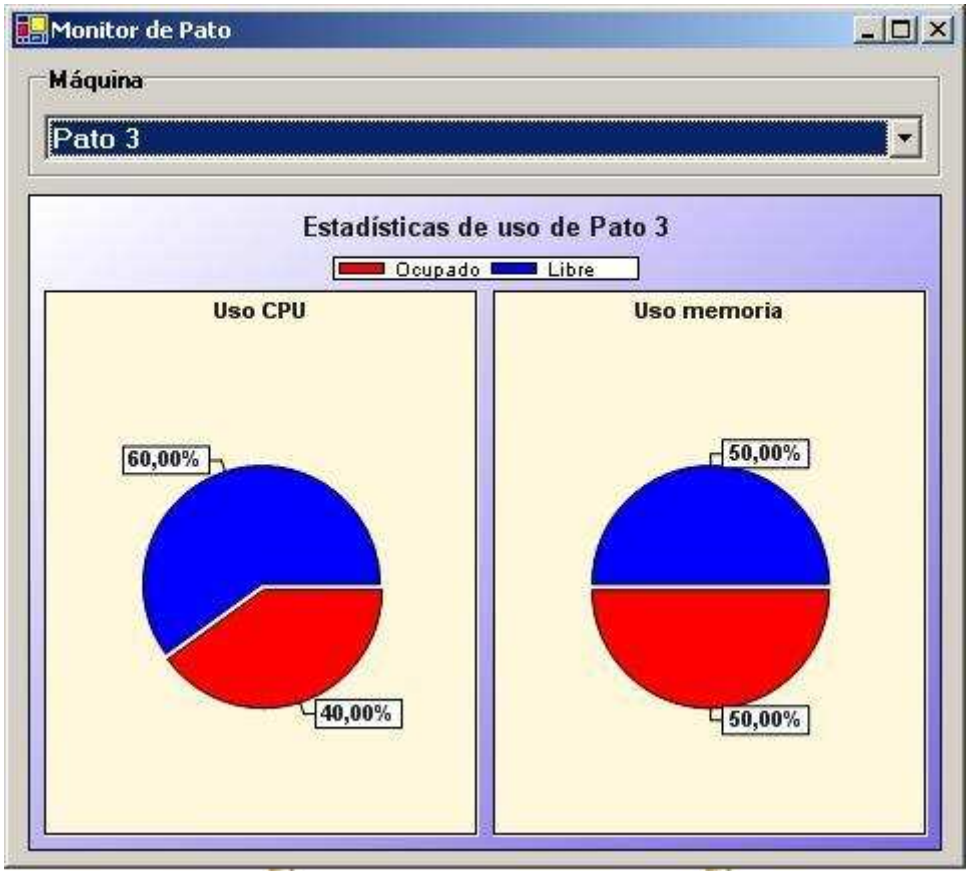
La representación gráfica escogida para representar los porcentajes ha sido el diagrama de tarta. Al disponer el cluster de ocho máquinas se ha optado por mostrar las estadísticas de una sola máquina en la pantalla. Para que el usuario pueda seleccionar la máquina de la que desea ver las estadísticas se ha incluido un *combobox*:



Prototipo de interfaz de usuario.

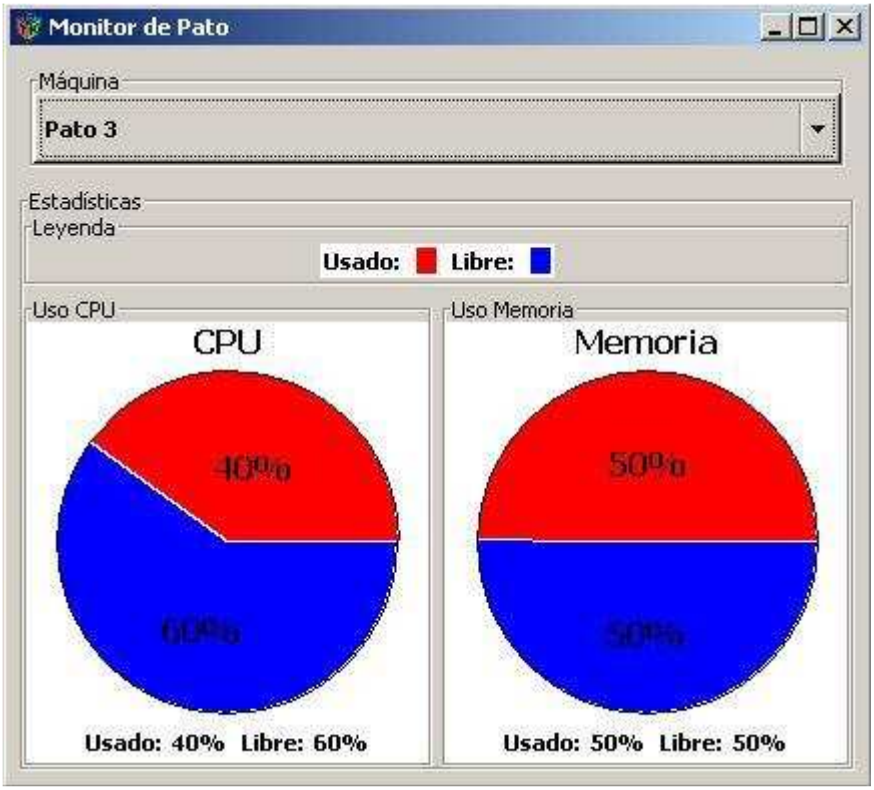
Basándose en el anterior prototipo hemos desarrollado los siguientes interfaces gráficos usando Java™ para las aplicaciones realizadas en Java™ y C# para las aplicaciones realizadas en .Net y Mono:

Toolkit	**System.Windows.Forms**
Descripción	Es el *toolkit* gráfico incluido por Microsoft® en la plataforma .Net sobre los controles estándar de Windows®.
Plataformas desarrollo	.Net.
Sistemas operativos	Windows®.
Ventajas	Existen editores de formularios de System.Windows.Forms incluidos en los entornos de desarrollo Visual Studio®; Borland® Galileo, usado por C#Builder y Delphi®, y #develop.
Inconvenientes	Solamente existen implementaciones estables para .Net, aunque los proyectos Mono y Portable.Net están desarrollando implementaciones multiplataformas para sus plataformas.
Notas	Para hacer los gráficos de tarta se ha usado el componente *ZedGraphControl* del proyecto ZedGraph distribuido bajo la licencia LGPL.



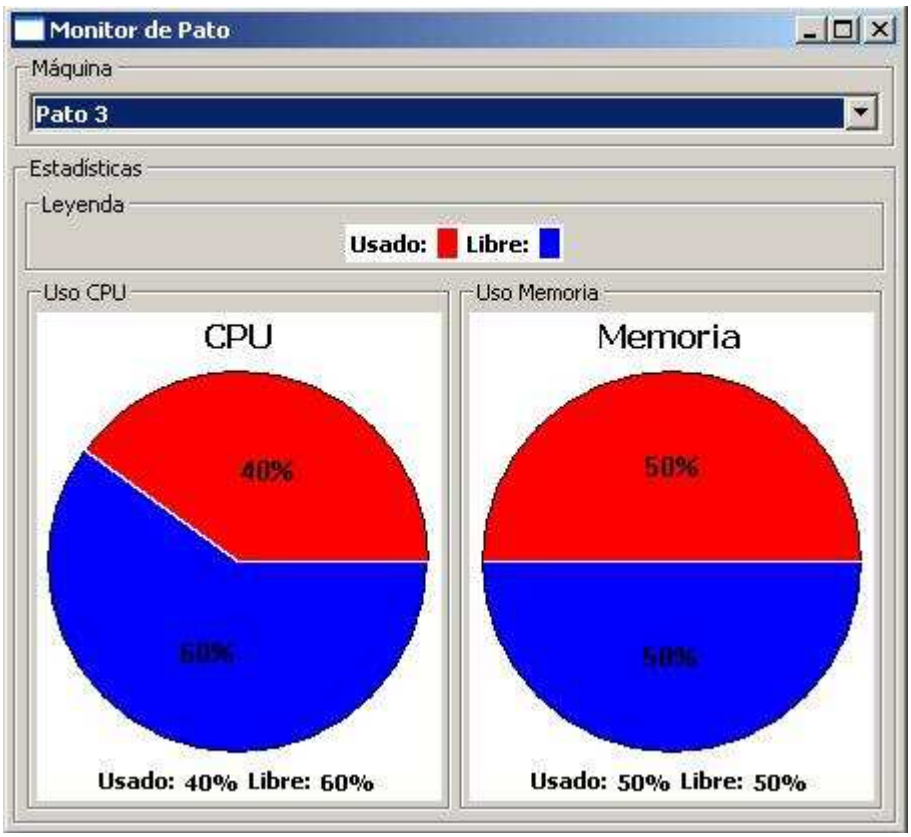
Interfaz System.Windows.Forms.

Toolkit	**GTK#**
Descripción	Es envoltorio escrito en C# con una pequeña capa de C del *toolkit* gráfico GTK+ y de varías de las librerías.
Plataformas desarrollo	.Net, Mono y Portable.Net.
Sistemas operativos	Windows®, Linux® y MacosX
Ventajas	Existen instaladores de GTK# para Windows® que automatizan todo el proceso de instalación del *toolkit* en el sistema operativo incluido también GTK+ y Glade. Todas las distribuciones Linux® que soportan Mono también soportan GTK#.
Inconvenientes	Se requiere que el usuario instale en su equipo el *toolkit* gráfico GTK+ tanto en Windows® como en Linux®.
Notas	En el caso de estar instalado Visual Studio® en el equipo en Windows® el instalador añadirá *templates* para desarrollar proyectos con GTK# y este entorno de desarrollo.



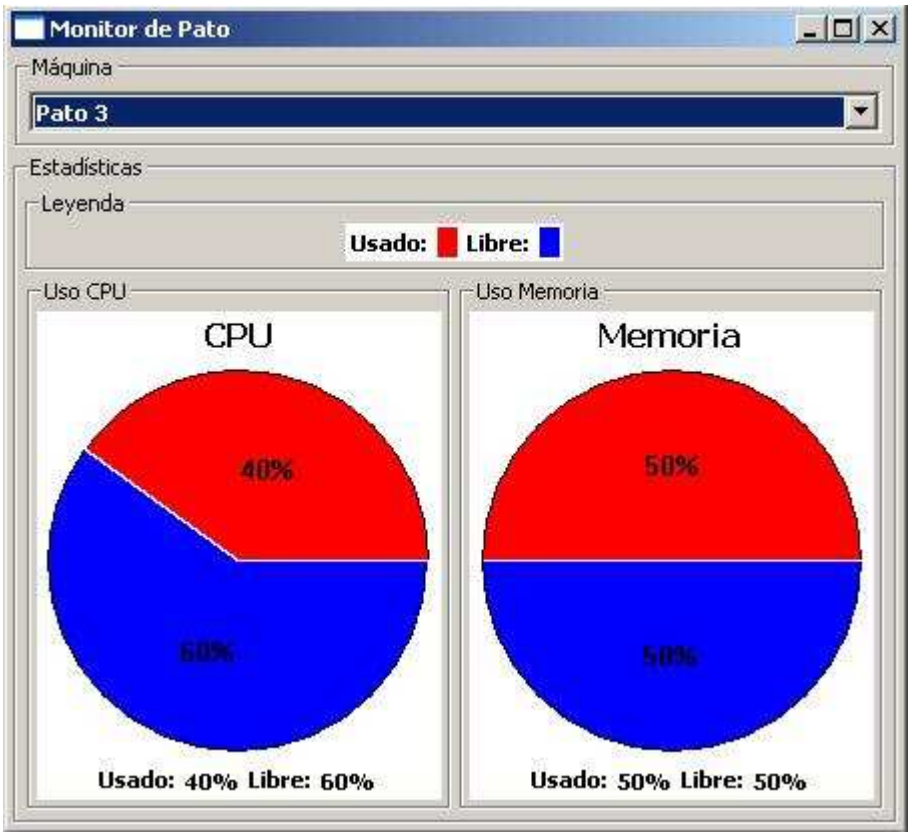
Interfaz GTK#.

Toolkit	**Wx.Net**
Descripción	Es envoltorio escrito en C# y C sobre el *toolkit* gráfico WxWidgets.
Plataformas desarrollo	.Net, Mono y Portable.Net.
Sistemas operativos	Windows®, Linux® y MacosX.
Ventajas	En Windows® solamente son necesarias dos .dlls, una la librería manipulada que contiene las clases que componen el *toolkit* y otra con la implementación de WxWidgets sobre GDI y el envoltorio en C de la misma.
Inconvenientes	En Linux® se requiere la instalación del *toolkit* gráfico GTK+.
Notas	El *toolkit* WxWidget es también un envoltorio en C++ de otros *toolkits* gráficos. La idea sobre la que está hecha WxWidgets es la de usar siempre los componentes nativos de la plataformas así en por ejemplo en Windows® usa Win32/GDI, en Linux® GTK+ o Motif y en Mac Carbon®.



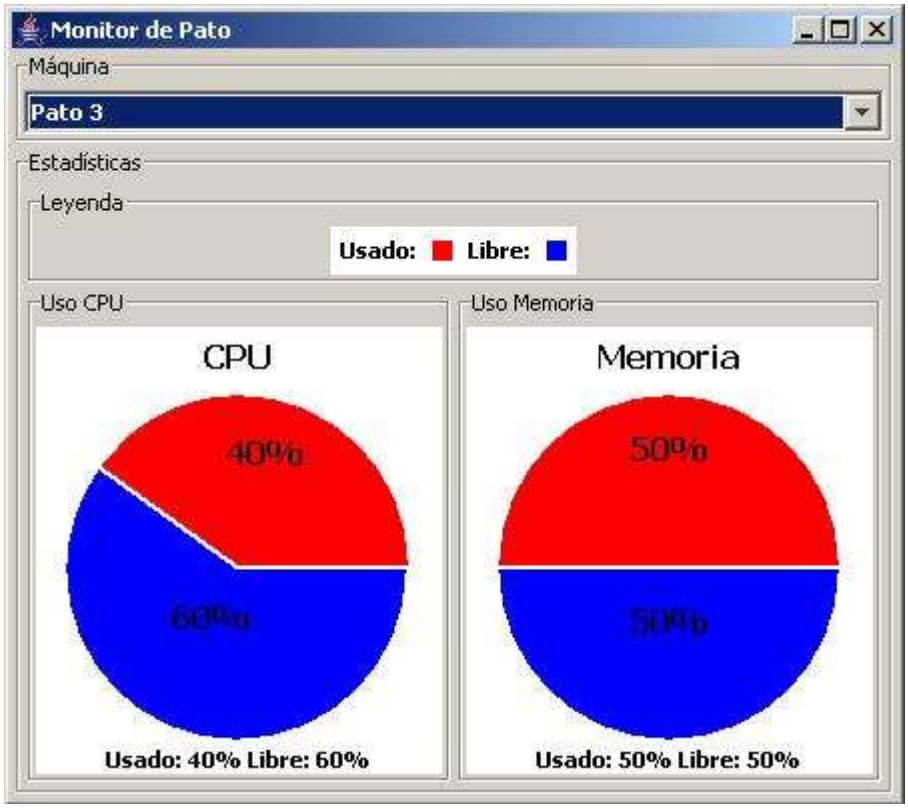
Interfaz Wx.Net.

Toolkit	**SWT y IKVM**
Descripción	SWT es el *toolkit* gráfico de Eclipse, está escrito en Java™ y C. SWT se puede usar en Mono y .Net compilando el archivo swt.jar a una .dll con la herramienta ikvmc de la máquina virtual de Java™ IKVM.
Plataformas desarrollo	.Net y Mono.
Sistemas operativos	Windows®, Linux® y MacosX.
Ventajas	Existe mucha documentación escrita sobre SWT con Java™ que podemos usar para desarrollar con C#, Visual Basic o cualquier otro lenguaje soportado por Mono y .Net.
Inconvenientes	En Linux® se requiere la instalación del *toolkit* gráfico GTK+. El archivo swt.jar es distinto cada sistema operativo por lo que la dll obtenidas a partir de él también.
Notas	La idea sobre la que está hecha SWT es la de usar siempre los componentes nativos de la plataformas así en por ejemplo en Windows® usa Win32/GDI, en Linux® GTK+ o Motif y en Mac Carbon®.



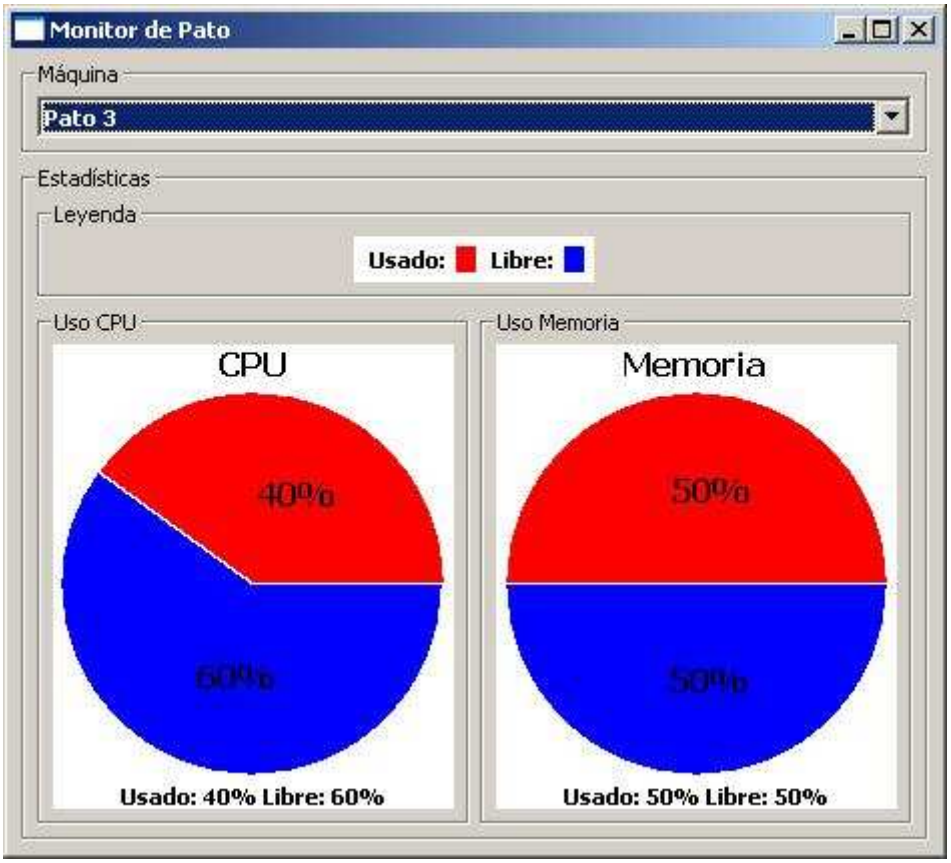
Interfaz SWT-IKVM.

Toolkit	**Swing**
Descripción	Swing es junto a AWT uno de los dos *toolkits* gráficos incluidos con la máquina virtual Java™ estándar desde la versión 1.2.
Plataformas desarrollo	Java™ Standard Edition v1.2 y siguientes.
Sistemas operativos	Windows®, Linux® y MacosX.
Ventajas	Al pertenecer a la versión estándar de Java™ existe mucha documentación sobre Swing. Existen numerosos editores de formularios para Swing como por ejemplo NetBeans Matisse o Eclipse Visual Editor.
Inconvenientes	No existen versiones no propietarias de Swing completamente funcionales.
Notas	Los componentes Swing están escritos en Java™ usando Java2D, parte del *toolkit* nativo AWT, como lienzo. Aunque la implementación de Swing es la misma en todas las plataformas existe un soporte para temas que hacen posible desarrollar interfaces con el aspecto de aplicaciones nativas.



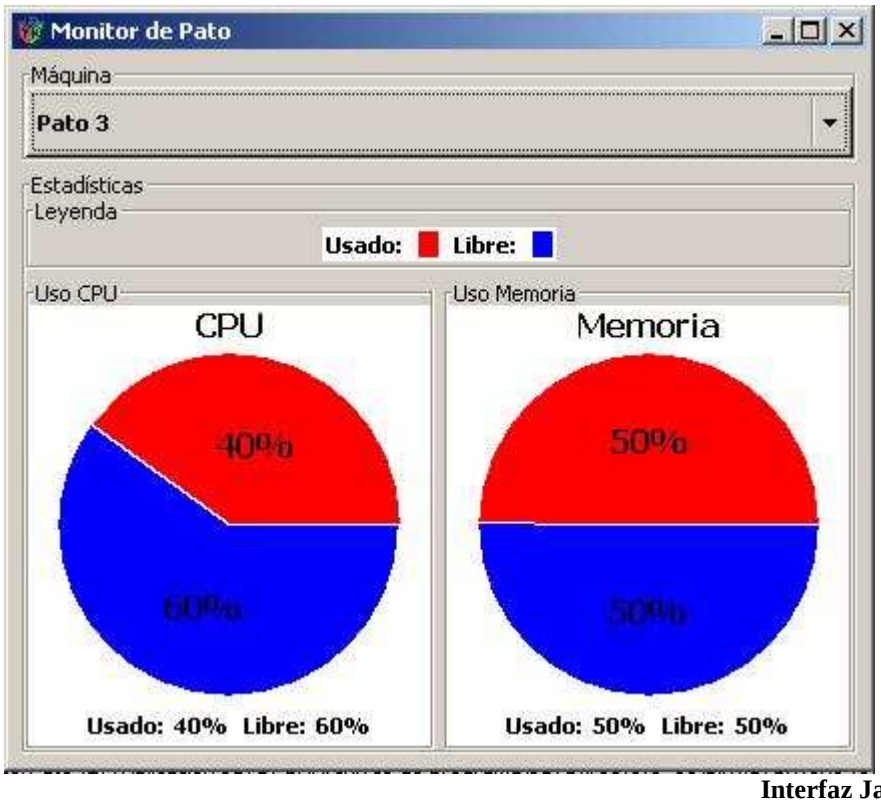
Interfaz Swing.

Toolkit	**SWT**
Descripción	SWT es el *toolkit* gráfico de Eclipse, está escrito en Java™ y C.
Plataformas desarrollo	Java™.
Sistemas operativos	Windows®, Linux® y MacosX.
Ventajas	Existe mucha documentación escrita sobre SWT, además SWT funciona en cualquier máquina virtual de Java™ que soporte JNI.
Inconvenientes	El archivo swt.jar en el que se distribuye SWT es diferente en cada sistema operativo, por lo que hemos de hacer varías distribuciones de nuestras aplicaciones.
Notas	La idea sobre la que está hecha SWT es la de usar siempre los componentes nativos de la plataformas así en por ejemplo en Windows® usa Win32/GDI, en Linux® GTK+ o Motif y en Mac Carbon®.



Interfaz SWT.

Toolkit	**JavaGnome**
Descripción	JavaGnome es un envoltorio de varías librerías del escritorio Gnome entre las que está el toolkit gráfico GTK+.
Plataformas desarrollo	Java
Sistemas operativos	Windows®, Linux® y MacosX.
Ventajas	Nos permite desarrollar aplicaciones para el escritorio Gnome con el lenguaje Java™. Funciona en cualquier máquina virtual de Java™ que soporte JNI.
Inconvenientes	En Windows® no podemos usar todas las librerías del escritorio Gnome. Debemos tener instalado el *toolkit* gráfico GTK+ en Windows®.
Notas	JavaGnome se ha desarrollado para poder realizar aplicaciones para el escritorio Gnome con Java, si solamente vamos a usar GTK+ resulta más lógico trabajar con SWT, que además de usar GTK+ en Linux® también ha sido pensada para realizar aplicaciones *portables*.



Interfaz JavaGnome.

6.5. Manual de usuario.

La herramienta de monitorización que acabamos de realizar al basarse en el servicio de eventos de CORBA™ tiene que arrancarse de acuerdo a la documentación de dicho servicio que está recogida en el apartado 4.1.1.6. .

Hemos de comenzar arrancando un servidor de nombres CORBA™, en el caso de Java™ se arranca con el siguiente comando:

```
orbd -ORBInitialPort 3000
```

A continuación arrancaremos el canal de eventos, ya sea la versión en Java™ o en Mono con uno de los comandos siguientes:

```
java -cp  jacorb.jar:. EventChannel -ORBInitialPort 3000
                                    –ORBInitialHost localhost

mono EventChannel -ORBInitialPort 3000
                                    –ORBInitialHost localhost
```

A partir de aquí el orden en que arranquemos los interfaces de usuario o los proveedores de eventos, comenzaremos por los proveedores:

```
java -cp  jacorb.jar:. ProveedorEventos -ORBInitialPort 3000
                  –ORBInitialHost pato1 –identificadorPato 1
                                    -periodoMilisegundos 1000


mono ProveedorEventos -ORBInitialPort 3000
                  –ORBInitialHost pato1 –identificadorPato 1
                                    -periodoMilisegundos 1000
```

A continuación solamente queda que los usuarios arranquen sus interfaces gráficos desde sus máquinas, aunque la manera de arrancar varía un poco con Java™ en función de los .jar requeridos un comando de arranque con los parámetros mínimos necesarios para todos los *toolkits* según la plataforma Java™ o .Net sería:

```
java -cp  jacorb.jar;. MonitorPato -ORBInitialPort 3000
                                    –ORBInitialHost pato1


MonitorPato -ORBInitialPort 3000 –ORBInitialHost pato1
```

7. Conclusiones y principales aportaciones.

El objetivo del proyecto, como vimos en la introducción, era obtener un catálogo se soluciones de interoperatividad entre las plataformas de desarrollo Java™ y CLI. Como el conjunto de soluciones era muy amplio se había decidido que aquellas que formaran parte del catálogo deberían tener una licencia de fuente abierta que no obligara a distribuir las aplicaciones que las usasen bajo una determinada licencia o que fueran parte de JRE™, Mono o el .Net Framework.

De la elaboración del catálogo de soluciones de interoperatividad hemos obtenido las cuatro siguientes conclusiones:

- Simplemente optar por *middleware* basado en estándares puede ser suficiente para la comunicación entre herramientas heterogéneas. Por ejemplo una herramienta Java™ que opta por usar RMI/IIOP en lugar de RMI normal, es decir RMI sobre JRMP, pueda comunicarse con programas escritos para una plataforma CLI con .Net Remoting y el canal IIOP.Net en lugar de TCPChannel.

- El realizar aplicaciones contemplando la interoperatividad no tiene porque ser más complicado. El uso del canal IIOP.Net que utiliza IIOP para las comunicaciones con .Net Remoting no es más difícil que usar el canal TCPChannel. Usar RMI/IIOP en lugar de RMI/JRMP no añade más complicación a la aplicación.

- Contemplar la interoperatividad en aplicaciones nuevas permite obtener mejores diseños. La gran apuesta de Microsoft® y SUN™ para la comunicación entre herramientas de la plataforma .Net y desarrollos en la plataforma Java™. Con los servicios web el desarrollador no tiene solamente un *middleware* estándar, también tiene un *middleware* basado en la arquitectura SOA. El acoplamiento con SOA es muy pequeño, el bajo acoplamiento entre los elementos de una aplicación permite sustituir un componente por otro más moderno sin tener que modificar el resto.

- La comunicación de aplicaciones heterogéneas se puede lograr a través agentes de mensajería. El agente de mensajería ActiveMQ puede enviar información entre dos aplicaciones, una escrita por ejemplo en Java™ con un cliente JMS propio de ActiveMQ y otra desarrollada con una plataforma CLI y la librería #STOMP.

Como ya hemos dicho, en este proyecto se pretendía elaborar u un catálogo de soluciones de interoperatividad, con la elaboración de dicho catalogo se han realizado las siguientes aportaciones:

- Conjunto ordenado de soluciones de interoperatividad. Las soluciones se han presentado desde las más sencillas a las más elaboradas. Se ha comenzado por aquellas en las que se consigue la interoperatividad a través de llamadas a método dentro de un mismo proceso hasta las más complejas en las que median programas

intermedios como agentes de mensajería o buses de integración empresariales.

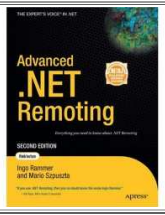
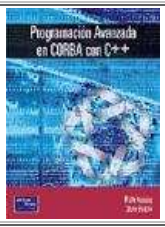
- Ejemplos sencillos de cada una de las soluciones de interoperatividad comentadas. Para cada solución planteada se ha realizado un ejemplo sencillo, tipo *hola Mundo*, con el fin de tratas la infraestructura mínima que requiere cada solución. Se ha tratado de que el catálogo sea útil para elegir entre soluciones, no un manual completo de cada una.
- Implementación del servicio de eventos de CORBA™. Se pretendía demostrar como la implementación de un servicio CORBA™ en C# era válida para trabajar con otra implementación ya existente en Java™.
- Librerías en C# para comunicar con agentes de mensajería. Se han desarrollado dos librerías para comunicarse con los agentes de mensajería Java™ ActiveMQ y XMLBlaster. Cualquier desarrollador que necesite realizar esta tarea en C# podrá utilizar nuestras implementaciones y usar el ejemplo del catálogo como manual de iniciación.
- Comparativa de rendimiento de dos implementaciones de canal de eventos de CORBA™. Cuando se realiza un nuevo desarrollo es conveniente compararlo con los ya existentes. Aquí hemos comparado nuestra implementación de canal de eventos en C# con aquella que hemos usado de partida. Al tratarse de implementaciones compatibles se han buscado entre varias combinaciones con las implementaciones de los roles del canal de eventos en ambas plataformas.
- Ejemplo de desarrollo con canal de eventos de una herramienta de monitorización de cluster. Se ha desarrollado una herramienta para monitorización de un cluster, el objetivo era tener un ejemplo de cómo adaptar un canal de eventos CORBA™ para la solución de un problema real. El lector ha podido ver como un único planteamiento ha servido para realizar las implementaciones en Java y C#. Como se ha partido de una solución estándar para cada rol del canal de eventos puede usarse una de las implementaciones y trabajar con las programas encargados de los otros roles con indiferencia de la plataforma sobre la que estén implementados estos.
- Pequeño catálogo de interfaces de usuario para las plataformas Java™, Mono y .Net Framework. Se han desarrollado varios interfaces de usuario para la herramienta de monitorización del cluster. Aunque no es un objetivo prioritario del proyecto aquí el lector también puede beneficiarse de un pequeño catálogo para escoger entre los *toolkits* de componentes gráficos para estas plataformas.

8. Trabajo futuro.

Este proyecto trata de servir como punto de partida de otros. Al estudiar varías de las soluciones de interoperatividad existentes nos hemos percatado de la no existencia de algunas implementaciones:

- Paso del estado de un objeto:
    - Implementación de un *formateador* XDR para las plataformas de desarrollo .Net Framework y Mono a partir del espacio de nombres *System.Reflection* implementando el interfaz *System.Runtime.Serialization.IFormatter*.
- Llamada a procedimiento remoto:
    - Implementar el protocolo RPC en C# a partir de la implementación RemoteTea en Java™. Esta implementación también implementaría un *formateador* XDR, aunque no siguiendo los estándares de CLI.
    - Implementar el servicio de nombres de CORBA™. En todos los ejemplos con CORBA™ de este proyecto se ha usado un servidor de nombres Java™. IIOP.Net solamente implementa la parte cliente del servicio de nombres. Para este desarrollo también se puede partir también se podría partir de la implementación de Jacorb.
- Paso de mensajes:
    - Extensión de la librería #STOMP para soportar los modelos publicación/suscripción y punto a punto. Se podría imitar el método con prefijos que usa ActiveMQ para distinguir entre colas y tópicos. En el caso del modelo punto a punto también se podrían implementar estrategias de balanceado para escoger el consumidor que recibe el evento cada vez.
    - Implementar un modelo de componentes para un ESB sobre Mono usando el *framework* Spring.Net, versión en C# del *framework* Spring que utilizan ServiceMix y ActiveMQ entre otros programas Java™.

Apéndice A) Bibliografía.

Portada	Título	Autores
	Advanced .NET Remoting, Second Edition. Apress © 2005	Ingo Rammer, Mario Szpuszta
	Inside COM. Microsoft Press © 1996	Dale Rogerson
	Programación avanzada en Corba con C++. Addison Wesley © 2002	Michi Henning, Steve Vinoski
	Swing Manning © 1999	Matthew Robinson, Pavel Vorobiev
	The Definitive Guide to SWT and Jface Apress © 2004	Robert Harris, Rob Warner

Apéndice B) Direcciones de Internet.

Referencia	Dirección
#Develop	http://www.icsharpcode.net/OpenSource/SD/
.Net Framework	http://www.microsoft.com/net/
ActiveMQ	http://www.activemq.org/
Ado.Net	http://msdn.microsoft.com/data/
Apache	http://www.apache.org/
Apache XMLRPC	http://ws.apache.org/xmlrpc/
Axis	http://ws.apache.org/axis/
Bea®	http://www.bea.com/
Borland®	http://www.borland.com/
C#	http://www.ecma-international.org/publications/standards/Ecma-334.htm
Caffeine.Net	http://caffeine.berlios.de/
Carbon®	http://developer.apple.com/carbon/
Cassini	http://www.asp.net/Projects/Cassini/Download/
Caucho	http://www.caucho.com/
CLI	http://www.ecma-international.org/publications/standards/Ecma-335.htm
Code::Blocks	http://www.codeblocks.org/
COM	http://www.microsoft.com/com/
COM4J	http://java.sun.com/j2se/1.4.2/docs/guide/jni/
Commons-logging	http://jakarta.apache.org/commons/logging/
Commons-math	http://jakarta.apache.org/commons/math/
CORBA®	http://www.corba.org/
CORBA® EventService	http://www.omg.org/technology/documents/formal/event_service.htm
db4o	http://www.db4o.com/
Delphi®	http://www.borland.com/delphi/
Eclipse	http://www.eclipse.org/
Eclipse Hibernate Synchronizer	http://hibernatesynch.sourceforge.net/
Eclipse JDT Batch Compiler	http://dev.eclipse.org/viewcvs/index.cgi/jdt-core-home/howto/batch%20compile/batchCompile.html?rev=1.5
Eclipse Visual Editor	http://www.eclipse.org/vep/
ECMA	http://www.ecma-international.org/
FatJar	http://fjep.sourceforge.net/
FreeBSD	http://www.freebsd.org/
Frontier	http://frontier.userland.com/

Referencia	Dirección
Gcc	http://gcc.gnu.org/
Gcj	http://gcc.gnu.org/java/
Glade	http://glade.gnome.org/
Glassfish	https://glassfish.dev.java.net/
GNOME	http://www.gnome.org/
GPL	http://www.gnu.org/copyleft/gpl.html
Groovy	http://groovy.codehaus.org/
GTK	http://www.GTK.org/
GTK#	http://www.mono-project.com/GTKSharp
Hessian	http://www.caucho.com/hessian/
Hessian#	http://www.hessiancsharp.org/
Hibernate	http://www.hibernate.org/
Ibatis	http://ibatis.apache.org/
IBM®	http://www.ibm.com/
IIOP.Net	http://iiop-net.sourceforge.net/
IKVM	http://www.ikvm.net/
J2EE™	http://java.sun.com/javaee/
Jackcess	http://jackcess.sourceforge.net/
JACOB	http://danadler.com/jacob/
Jacorb	http://www.jacorb.org/
Janeva™	http://www.borland.com/us/products/janeva/
Java™	http://java.sun.com/
JavaGnome	http://java-gnome.sourceforge.net/
JBI	http://www.jcp.org/en/jsr/detail?id=208
Jboss™	http://www.jboss.com/
JCP™	http://www.jcp.org/
JDBC™	http://java.sun.com/products/jdbc/
Jetty®	http://jetty.mortbay.org/jetty/
J-Integra™	http://j-integra.intrinsyc.com/
JMS	http://java.sun.com/products/jms/
JNDI™	http://java.sun.com/products/jndi/
JNI	http://java.sun.com/j2se/1.4.2/docs/guide/jni/
JRuby	http://jruby.sourceforge.net/
JuggerNet™	http://www.codemesh.com/en/JuggerNETProduct_page.html
Jython	http://www.jython.org/
LGPL	http://www.gnu.org/copyleft/lesser.html
LINQ	http://msdn.microsoft.com/netframework/future/linq/
Linux®	http://www.linux.org/
Matisse	http://www.netbeans.org/kb/articles/matisse.html
Microsoft®	http://www.microsoft.com/
Middcor.Net™	http://www.middsol.com/middcorcorba.html

Referencia	Dirección
MinGW	http://www.mingw.org/
MIT license	http://www.opensource.org/licenses/mit-license.php
Mono	http://www.mono-project.com/
Monodevelop	http://www.monodevelop.org/
Motif	http://www.opengroup.org/motif/
Mozilla®	http://www.mozilla.org/
MSDN	http://msdn.microsoft.com/
MSMQ	http://www.microsoft.com/msmq/
MSMQJava	http://blogs.msdn.com/dotnetinterop/archive/2005/02/28/381735.aspx
Mule	http://mule.codehaus.org/
Nant	http://nant.sourceforge.net/
NetBeans	http://www.netbeans.org/
NHibernate	http://www.hibernate.org/
Novell®	http://www.novell.com/
OMG®	http://www.omg.org/
Oracle®	http://www.oracle.com/
Parrot	http://www.parrotcode.org/
Perl	http://www.perl.org/
PHP	http://www.php.net/
POI	http://jakarta.apache.org/poi/
Portable.Net	http://www.dotgnu.org/pnet.html
PostgreSQL	http://www.postgresql.org/
Python	http://www.python.org/
RemoteTea	http://remotetea.sourceforge.net/
Remoting.CORBA	http://remoting-corba.sourceforge.net/
Resin	http://www.caucho.com/resin/
RMI	http://java.sun.com/products/jdk/rmi/
RMI/IIOP	http://java.sun.com/products/rmi-iiop/
Ruby	http://www.ruby-lang.org/
Salamander.Net	http://www.remotesoft.com/salamander/
SerSTOMP	http://www.germane-software.com/software/Java/SERStomp/
ServiceMix	http://incubator.apache.org/servicemix/
SOAP	http://www.w3.org/TR/SOAP/
Spring Framework	http://www.springframework.org/
Spring.Net	http://www.springframework.net/
STOMP	http://stomp.codehaus.org/
Sun™	http://www.sun.com/
SWIG	http://www.swig.org/
SWT	http://www.eclipse.org/swt/
TCL	http://tcl.sourceforge.net/
W3C	http://www.w3.org/

Referencia	Dirección
WCF	http://msdn.microsoft.com/webservices/indigo/default.aspx
Windows®	http://www.microsoft.com/windows
WSE	http://msdn.microsoft.com/webservices/webservices/building/wse/default.aspx
Wx.Net	http://wxnet.sourceforge.net/
WxWidgets	http://www.wxwidgets.org/
XMLBeans	http://xmlbeans.apache.org/
XMLBlaster	http://www.xmlblaster.org/
XMLRPC.Net	http://www.xml-rpc.net/
XPath	http://www.w3.org/TR/XPath
XPCOM	http://www.mozilla.org/projects/xpcom/
XSLT	http://www.w3.org/TR/xslt
ZedGraph	http://zedgraph.sourceforge.net/